USENIX

conference

proceedings

3rd USENIX Conference on File and Storage Technologies

San Francisco, California, USA
March 31–April 2, 2004

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with ACM SIGOPS, IEEE Mass Storage Systems Technical Committee (MSSTC), and IEEE TCOS

Proceedings of the 3rd USENIX Conference on File and Storage Technologies *San Francisco, CA, Mar. 31–Apr. 2, 2004*

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $30 for members and $40 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-931971-19-6

USENIX Association

Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)

March 31–April 2, 2004
San Francisco, California, USA

Conference Organizers

Program Chair

Chandu Thekkath, *Microsoft Research*

Program Committee

Guillermo Alvarez, *IBM Almaden*
Fay Chang, *Google*
Jeff Chase, *Duke University*
Greg Ganger, *Carnegie Mellon University*
Richard Golding, *IBM Almaden*
Dirk Grunwald, *University of Colorado*
Chet Juszczak, *Sun Microsystems*
Christos Karamanolis, *Hewlett-Packard Labs*
Ed Lee, *Data Domain*
David Patterson, *University of California, Berkeley*
Randy Wang, *Princeton University*
Yuanyuan Zhou, *University of Illinois at Urbana-Champaign*

Steering Committee

Jeff Chase, *Duke University*
Jack Cole, *US Army*
Greg Ganger, *Carnegie Mellon University*
Garth Gibson, *Panasas and Carnegie Mellon University*
Peter Honeyman, *CITI, University of Michigan*
John Howard, *Sun Microsystems*
Merritt Jones, *MITRE Corporation*
Darrell Long, *University of California, Santa Cruz*
Jai Menon, *IBM Research*
Margo Seltzer, *Harvard University*
John Wilkes, *Hewlett-Packard Labs*
Ellie Young, *USENIX*

The USENIX Association Staff

External Reviewers

Anurag Acharya
Atul Adya
Marcos Aguilera
Stephen Aiken
Khalil Amiri
Stergios Anastasiadis
Darrell Anderson
Eric Anderson
Paul Barham
Ralph Becker-Szendy
Angelos Bilas
David Blacklock
Chris Boire
Bill Bolosky
Scott Brandt
Walt Burkhard
Mike Burrows
Pei Cao
Miguel Castro
Peter Chen
Zhifeng Chen
Peter Corbett
Dan Crawl
Erick Crowell
Mike Dahlin
Daniel Ellard
Stephen Ferrari
Nitin Garg
Sanjay Ghemawat
Howard Gobioff
Garth Goodson
Alex Halderman
Roger Haskin
John Howard
Terril Hurst
Larry Huston
Rebecca Isaacs
Michael Isard
Minwen Ji
MacCormick John
William Josephson
Ajdan Jumerefendi
Mahesh Kallahalla
Magnus Karlsson
Kimberly Keeton
Terence Kelly
Deepak Kenchammana-Hosekote
Sanjeev Kumar
Junwen Lai
Darrell Long
Mike Luby
Stan Luke
Christopher R. Lumb
Marty Lund
Qin Lv
John MacCormick
Umesh Maheshwari
Mark Manasse
David Martin
David Mazières
Richard McDougall
Kirk McKusick
Arif Merchant
Milan Merhar
Ethan Miller
Justin Moore
Charles Morrey
Robert Morris
David Nagle
Brian Noble
Prashant Pandey
Nitin Parab
Sharon Perl
Rob Pike
Dulce Ponceleon
Ron Proulx
Sean Quinlan
Aseem Rastogi
Sazzala Reddy
Erik Riedel
Rick Roche
Yasushi Saito
Mahadev Satyanarayanan
Jiri Schindler
Michael Schmitz
Sid Selkirk
Yilei Shao
Piyush Shivam
Pratap Singh
Harald Skardal
Sumeet Sobti
Craig Soules
Sudarshan Srinivasan
Steve Strange
Florin Sultan
Chait Tumuluri
Mustafa Uysal
Alistair Veitch
Tirunelveli Vishwanath
Limin Wang
Brent Waters
Rajiv Wickremesinghe
John Wilkes
Ted Wobber
Matthew Woitaszek
Jay Wylie
Zhichen Xu
Lawrence You
Xiang Yu
Jeffrey Zabarsky
Erez Zadok
Chi Zhang
Lidong Zhou
Ben Zhu

FAST '04: 3rd USENIX Conference on File and Storage Technologies

March 31–April 2, 2004
San Francisco, CA, USA

Wednesday, March 31, 2004

Reliability & Availability
Session Chair: Chandu Thekkath, Microsoft Research

Measurement, Modeling, and Management
Session Chair: Richard Golding, IBM Almaden

Thursday, April 1, 2004

Grabbag
Session Chair: Jeff Chase, Duke University

File Systems
Session Chair: Greg Ganger, Carnegie Mellon University

Optimizing Block Access
Session Chair: Guillermo Alvarez, IBM Almaden

Friday, April 2, 2004

Caching & Scheduling
Session Chair: Randy Wang, Princeton University

Mobile Storage
Session Chair: Dirk Grunwald, University of Colorado

Index of Authors

Message from the Program Chair

I am delighted to welcome you to the 3rd USENIX Conference on File and Storage Technologies (FAST '04). FAST '04 is a single-track conference that brings together some of the best papers in the area of storage and file systems. The proceedings contain 18 papers spanning a wide spectrum of topics written by researchers from academia and industry.

This year we received 72 papers, slightly higher than last year, but lower than the inaugural year.

Papers were reviewed in two stages. In the first stage, each paper received three reviews from program committee members and two external reviews. In the second stage, the top half of these papers received at least two additional reviews from the program committee. This led to a very informed and lively committee meeting. Each of the accepted papers was shepherded by a program committee member familiar with the topic.

We were fortunate to have a superb and dedicated committee drawn from academia and industry who worked tremendously hard to produce the program you see before you. It was a privilege serving on the committee, and I would like to thank all the members for their efforts.

To process the large number of reviews, we chose to use Dirk Grunwald's paper reviewing system from the University of Colorado. I would like to thank Dirk for taking the time to maintain this software, which worked flawlessly for all intents and purposes.

I would also like to thank all the authors who submitted papers to FAST and the external reviewers for all their hard work. It is also a pleasure to acknowledge the help from the USENIX staff, especially Ellie Young and Jane-Ellen Long, for organizing the conference. Thanks are also due to Johnny Foehner and Claudia Boyle of Microsoft for their able administrative help.

Chandramohan A. Thekkath, ***Microsoft Research***
Program Chair

Row-Diagonal Parity for Double Disk Failure Correction

Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac,
Steven Kleiman, James Leong, and Sunitha Sankar
Network Appliance, Inc.

Abstract

Row-Diagonal Parity (RDP) is a new algorithm for protecting against double disk failures. It stores all data unencoded, and uses only exclusive-or operations to compute parity. RDP is provably optimal in computational complexity, both during construction and reconstruction. Like other algorithms, it is optimal in the amount of redundant information stored and accessed. RDP works within a single stripe of blocks of sizes normally used by file systems, databases and disk arrays. It can be utilized in a fixed (RAID-4) or rotated (RAID-5) parity placement style. It is possible to extend the algorithm to encompass multiple RAID-4 or RAID-5 disk arrays in a single RDP disk array. It is possible to add disks to an existing RDP array without recalculating parity or moving data. Implementation results show that RDP performance can be made nearly equal to single parity RAID-4 and RAID-5 performance.

1 Introduction

Disk striping techniques [1, 2] have been used for more than two decades to reduce data loss due to disk failure, while improving performance. The commonly used RAID techniques, RAID-4 and RAID-5, protect against only a single disk failure. Among the standard RAID techniques, only mirrored stripes (RAID-10, RAID-01) provide protection against multiple failures. However, they do not protect against double disk failures of opposing disks in the mirror. Mirrored RAID-4 and RAID-5 protect against higher order failures [4]. However, the efficiency of the array as measured by its data capacity divided by its total disk space is reduced. Increasing the redundancy by small increments per stripe is more cost effective than adding redundancy by replicating the entire array [3].

The dramatic increase in disk sizes, the relatively slower growth in disk bandwidth, the construction of disk arrays containing larger numbers of disks, and the use of less reliable and less performant varieties of disk such as ATA combines to increase the rate of double disk failures, as will be discussed in Section 3. This requires the use of algorithms that can protect against double disk failures to ensure adequate data integrity. Algorithms that meet information theory's Singleton bound [6] protect against two disk failures by adding only two disks of redundancy to the number of disks required to store the unprotected data. Good algorithms meet this bound, and also store the data unencoded, so that it can be read directly off disk.

A multiple orders of magnitude improvement in the reliability of the storage system can simplify the design of other parts of the system for robustness, while improving overall system reliability. This motivates the use of a data protection algorithm that protects against double disk failures. At the same time, it is desirable to maintain the simplicity and performance of RAID-4 and RAID-5 single parity protection.

This paper describes a new algorithm, called Row-Diagonal Parity, or RDP, for protection against double failures. RDP applies to any multiple device storage system, or even to communication systems. In this paper, we focus on the application of RDP to disk array storage systems (RAID).

RDP is optimal both in computation and in I/O. It stores user data in the clear, and

requires exactly two parity disks. It utilizes only exclusive-or operations during parity construction as well as during reconstruction after one or two failures. Therefore, it can be implemented easily either in dedicated hardware, or on standard microprocessors. It is also simple to implement compared to previous algorithms. While it is difficult to measure the benefit of this, we were able to implement the algorithm and integrate it into an existing RAID framework within a short product development cycle.

In this paper, we make the case that the need for double disk failure protection is increasing. We then describe the RDP algorithm, proving its correctness and analysing its performance. We present some simple extensions to the algorithm, showing how to add disks to an existing array, and how to protect multiple RAID-4 or RAID-5 arrays against double failures with a single extra parity disk. Finally, we present some observations from our experience implementing RDP, and give some performance results for that implementation.

2 Related Work

There are several known algorithms that protect data against two or more disk failures in an array of disks. Among these are EVENODD [5], Reed Solomon (P+Q) erasure codes [6], DATUM [7] and RM2 [8]. RDP is most similar to EVENODD. RM2 distributes parity among the disks in a single stripe, or equivalently, adds stripes of parity data that are interspersed among the data stripes. EVENODD, DATUM, and Reed-Solomon P+Q all share the property that the redundant information can be stored separately from the data in each stripe. This allows implementations that have dedicated redundant disks, leaving the other disks to hold only data. This is analogous to RAID-4, although we have two parity disks, not one. We will call this RAID-4 style parity placement. Alternatively, the placement of the redundant information can be rotated from stripe to stripe, improving both read and write performance. We will call this RAID-5 style parity placement.

Both EVENODD and Reed-Solomon P+Q encoding compute normal row parity for one parity disk. However, they employ different techniques for encoding the second disk of redundant data. Both use exclusive-or operations, but Reed-Solomon encoding is much more computationally intensive than EVENODD [5]. DATUM uses encodings that generate any number of redundant information blocks. It allows higher order failure tolerance, and is similar to Reed-Solomon P+Q encoding in the case of protection against two disk failures.

RDP shares many of the properties of EVENODD, DATUM, and Reed-Solomon encoding, in that it stores its redundant data (parity) separately on just two disks, and that data is stored in the clear on the other disks. Among the previously reported algorithms, EVENODD has the lowest computational cost for protection against two disk failures. RDP improves upon EVENODD by further reducing the computational complexity. The complexity of RDP is provably optimal, both during construction and reconstruction. Optimality of construction is important as it is the normal, failure free operational mode. However, the optimality of reconstruction is just as important, as it maximizes the array's performance under degraded failure conditions [9].

3 Double Disk Failure Modes and Analysis

Double disk failures result from any combination of two different types of single disk failure. Individual disks can fail by whole-disk failure, whereby all the data on the disk becomes temporarily or permanently inaccessible, or by media failure, whereby a small portion of the data on a disk becomes temporarily or permanently inaccessible. Whole-disk failures may result from a problem in the disk itself, or in the channel or network connecting the disk to its containing system. While the mode and duration of the failures may vary, the class of failures that make the

data on a disk unaccessible can be categorized as one failure type for the purposes of recovery. Whole-disk failures require the complete reconstruction of a lost disk, or at least those portions of it that contain wanted data. This stresses the I/O system of the controller, while adding to its CPU load. (We will refer to the unit that performs construction of parity and reconstruction of data and parity as the *controller*.)

To maintain uninterrupted service, the controller has to serve requests to the lost disk by reconstructing the requested data on demand. At the same time, it will reconstruct the other lost data. It is desirable during reconstruction to have a low response time for the on-demand reconstruction of individual blocks that are required to service reads, while at the same time exhibiting a high throughput on the total disk reconstruction.

Whole-disk failure rates are measured as an arrival rate, regardless of the usage pattern of the disk. The assumption is that the disk can go bad at any time, and that once it does, the failure will be noticed. Whole disk failure rates are the reciprocal of the Mean Time To Failure numbers quoted by the manufacturers. These are typically in the range of 500,000 hours.

Media failures are qualitatively and quantitatively different from whole-disk failures. Media failures are encountered during disk reads and writes. Media failures on write are handled immediately, either by the disk or by the controller, by relocating the bad block to a good area on disk. Media failures on read can result in data loss. While a media failure only affects a small amount of data, the loss of a single sector of critical data can compromise an entire system. Handling media failures on read requires a short duration recovery of a small amount of missing data. The emphasis in the recovery phase is on response time, but reconstruction throughput is generally not an issue.

Disks protect against media errors by relocating bad blocks, and by undergoing elaborate retry sequences to try to extract data from a sector that is difficult to read [10]. Despite these precautions, the typical media error rate in disks is specified by the manufacturers as one bit error per 10^{14} to 10^{15} bits read, which corresponds approximately to one uncorrectable error per 10TBytes to 100TBytes transferred. The actual rate depends on the disk construction. There is both a static and a dynamic aspect to this rate. It represents the rate at which unreadable sectors might be encountered during normal read activity. Sectors degrade over time, from a writable and readable state to an unreadable state.

A second failure can occur during reconstruction from a single whole-disk failure. At this point, the array is in a degraded mode, where reads of blocks on the failed disk must be satisfied by reconstructing data from the surviving disks, and commonly, where the contents of the failed disk are being reconstructed to spare space on one or more other disks. If we only protect against one disk failure, a second complete disk failure will make reconstruction of a portion of both lost disks impossible, corresponding to the portion of the first failed disk that has not yet been reconstructed. A media failure during reconstruction will make reconstruction of the two missing sectors or blocks in that stripe impossible. Unfortunately, the process of reconstruction requires that all surviving disks are read in their entirety. This stresses the array by exposing all latent media failures in the surviving disks.

The three double disk failure combinations are: whole-disk/whole-disk, whole-disk/media, and media/media. A properly implemented double failure protection algorithm protects against all three categories of double failures. In our analysis of failure rates, we discount media/media failures as being rare relative to the other two double failure modes. Whole-disk/whole-disk and whole-disk/media failures will normally be encountered during reconstruction from an already identified whole-disk failure.

RAID systems can protect against double failures due to media failures by periodically "scrubbing" their disks, trying to read each sector, and reconstructing and relocating data on any sector that is unreadable. Doing this before a single whole-disk failure oc-

curs can preempt potential whole-disk/media failures by cleansing the disks of accumulated media errors before a whole-disk failure occurs. Such preventive techniques are a necessary precaution in arrays of current large capacity disks.

The media and whole-disk failure rates assume uniform failure arrivals over the lifetime of the disk, and uniform failure arrival rates over the population of similar disks. Actual whole-disk failure rates conform to a bathtub curve as a function of the disk's service time: A higher failure rate is encountered during the beginning-of-life burn-in and end-of-life wear-out periods. Both of these higher rate periods affect the double disk failure rate, as the disks in an array will typically be the same age, and will be subject to the same usage pattern. This tends to increase the correlation of whole-disk failures among the disks in an array.

Disks in the array may be from the same manufacturing batch, and therefore may be subject to the same variations in manufacturing that can increase the likelihood of an individual disk failing. Disks in an array are all subject to the same temperature, humidity and mechanical vibration conditions. They may all have been subjected to the same mechanical shocks during transport. This can result in a clustering of failures that increases the double failure rate beyond what would be expected if individual disk failures were uncorrelated.

Once a single disk fails, the period of vulnerability to a second whole-disk failure is determined by the reconstruction time. In contrast, vulnerability to a media failure is fixed once the first disk fails. Reconstruction will require a complete read of all the surviving disks, and the probability of encountering a media failure in those scans is largely independent of the time taken by reconstruction.

If the failures are independent, and wide sense stationary [12], then it is possible to derive the rate of occurance of two whole-disk failures as [2]:

$$\lambda_2 \approx \lambda_1^2 t_r c \frac{n(n-1)}{2} \qquad (1)$$

where t_r is the reconstruction time of a failed disk, n is the total number of disks in the array, λ_1 is the whole-disk failure rate of one disk, and c is a term reflecting the correlation of the disk failures. If whole-disk failures are correlated, then the correction factor $c > 1$. We know from experience that whole-disk failures are not stationary, i.e., they depend on the service time of the disk, and also that they are positively correlated. These factors will increase the rate λ_2.

The other consideration is that the reconstruction time t_r is a function of the total data that must be processed during reconstruction. t_r is linearly related to the disk size, but also can be related to the number of disks, since the total data to be processed is the product dn, where d is the size of the disks. For small n, the I/O bandwidths of the individual disks will dominate reconstruction time. However, for large enough n, the aggregate bandwidth of the disks becomes great enough to saturate either the I/O or processing capacity of the controller performing reconstruction. Therefore, we assert that:

$$t_r = \begin{cases} d/b_r & \text{if } n < m \\ dn/b_s & n \geq m \end{cases} \qquad (2)$$
$$m = \left\lceil \frac{b_s}{b_r} \right\rceil$$

where b_r is the maximum rate of reconstruction of a failed disk, governed by the disk's write bandwidth and b_s is the maximum rate of reconstruction per disk array.

The result for disk arrays larger than m is:

$$\lambda_2 \approx \frac{\lambda_1^2 dc}{2b_s} n^2(n-1) \qquad (3)$$

The whole-disk/whole-disk failure rate has a cubic dependency on the number of disks in the array, and a linear dependency on the size of the disks. The double failure rate is related to the square of the whole-disk failure rate. If we employ disks that have higher failure rates, such as ATA drives, we can expect that the double failure rate will increase proportionally to the square of the increase in single disk failure rate.

As an example, if the primary failure rate is one in 500,000 hours, the correlation factor is 1, the reconstruction rate is $100MB/s$,

in a ten disk array of 240 GByte disks, the whole-disk/whole-disk failure rate will be approximately 1.2×10^{-9} failures per hour.

Both the size of disks and their I/O bandwidth have been increasing, but the trend over many years has been that disk size is increasing much faster than the disk media rate. The time it takes to read or write an entire disk is the lower bound on disk recovery. As a result, the recovery time per disk has been increasing, further aggravating the double disk failure rate.

The rate of whole-disk/media failures is also related to disk size and to the number of disks in the array. Essentially, it is the rate of single whole-disk failures, multiplied by the probability that any of those failures will result in a double failure due to the inability to read all sectors from all surviving disks. The single whole-disk failure rate is proportional to the number of disks in the array. The media failure rate is roughly proportional to the total number of bits in the surviving disks of the array. The probability of all bits being readable is $(1-p)^s$ where p is the probability of an individual bit being unreadable, and s is the number of bits being read. This gives the a priori rate of whole-disk/media double failures:

$$f_2 = \lambda_1 n(1-(1-p)^{(n-1)b}) \qquad (4)$$

where b is the size of each disk measured in bits.

For our example of a primary failure rate of 1 in 500,000 hours, a 10 disk array, 240 GB disks, and a bit error rate of 1 per 10^{14} gives a whole-disk/media double failure rate of 3.2×10^{-6} failures per hour.

In our example, using typical numbers, the rate of whole-disk/media failures dominates the rate of whole-disk/whole-disk failures. The incidence of media failures per whole-disk failure is uncomfortably high. Scrubbing the disks can help reduce this rate, but it remains a significant source of double disk failures.

The combination of the two double failure rates gives a Mean Time To Data Loss (MTTDL) of 3.1×10^5 hours. For our example, this converts to an annual rate of 0.028 data loss events per disk array per year due to double failures of any type.

To compare, the dominant triple failure mode will be media failures discovered during recovery from double whole-disk failures. This rate can be approximated by the analog to Equation 4:

$$f_3 = \lambda_2(1-(1-p)^{(n-2)b}) \qquad (5)$$

Substituting λ_2 from Equation 1 gives:

$$f_3 \approx \frac{\lambda_1^2 dc}{2b_s} n^2(n-1)(1-(1-p)^{(n-2)b}) \qquad (6)$$

For our example, the dominant component of the tertiary failure rate will be approximately 1.7×10^{-10} failures per hour, which is a reduction of over four orders of magnitude compared to the overall double failure rate.

The use of less expensive disks, such as ATA disks, in arrays where high data integrity is required has been increasing. The disks are known to be less performant and less reliable than SCSI and FCP disks [10]. This increases the reconstruction time and the individual disk failure rates, in turn increasing the double failure rate for arrays of the same size.

4 Row-Diagonal Parity Algorithm

The RDP algorithm is based on a simple parity encoding scheme using only exclusive-or operations. Each data block belongs to one row parity set and to one diagonal parity set. In the normal configuration, there is one row parity block and one diagonal parity block per stripe. It is possible to build either RAID-4 or RAID-5 style arrays using RDP, by either locating all the parity blocks on two disks, or by rotating parity from disk to disk in different stripes.

An RDP array is defined by a controlling parameter p, which must be a prime number greater than 2. In the simplest construction of an RDP array, there are $p+1$ disks. We

Data Disk 0	Data Disk 1	Data Disk 2	Data Disk 3	Row Parity	Diag. Parity
0	1	2	3	4	0
1	2	3	4	0	1
2	3	4	0	1	2
3	4	0	1	2	3

Figure 1: Diagonal Parity Set Assignments in a 6 Disk RDP Array, $p = 5$

define stripes across the array to consist of one block from each disk. In each stripe, one block holds diagonal parity, one block holds row parity, and $p - 1$ blocks hold data.

The bulk of the remainder of this paper describes one grouping of $p - 1$ stripes that includes a complete set of row and diagonal parity sets. Multiple of these stripe groupings can be concatenated to form either a RAID-4 style or RAID-5 style array. An extension to multiple row parity sets is discussed in Section 7.

Figure 1 shows the four stripes in a 6 disk RDP array ($p = 5$). The number in each block indicates the diagonal parity set the block belongs to. Each row parity block contains the even parity of the data blocks in that row, not including the diagonal parity block. Each diagonal parity block contains the even parity of the data and row parity blocks in the same diagonal. Note that there are $p = 5$ diagonals, but that we only store the parity of $p - 1 = 4$ of the diagonals. The selection of which diagonals to store parity for is completely arbitrary. We refer to the diagonal for which we do not store parity as the "missing" diagonal. In this paper, we always select diagonal $p - 1$ as the missing diagonal. Since we do not store the parity of the missing diagonal, we do not compute it either.

The operation of the algorithm can be seen by example. Assume that data disks 1 and 3 have failed in the array of Figure 1. It is necessary to reconstruct from the remaining data and parity disks. Clearly, row parity is useless in the first step, since we have lost two members of each row parity set. However, since each diagonal misses one disk, and all diagonals miss a different disk, then there are two diagonal parity sets that are only missing one block. At least one of these two diagonal parity sets has a stored parity block. In our example, we are missing only one block from each of the diagonal parity sets 0 and 2. This allows us to reconstruct those two missing blocks.

Having reconstructed those blocks, we can now use row parity to reconstruct two more missing blocks in the two rows where we reconstructed the two diagonal blocks: the block in diagonal 4 in data disk 3 and the block in diagonal 3 in data disk 1. Those blocks in turn are on two other diagonals: diagonals 4 and 3. We cannot use diagonal 4 for reconstruction, since we did not compute or store parity for diagonal 4. However, using diagonal 3, we can reconstruct the block in diagonal 3 in data disk 3. The next step is to reconstruct the block in diagonal 1 in data disk 1 using row parity, then the block in diagonal 1 in data disk 3, then finally the block in diagonal 4 in data disk 1, using row parity.

The important observation is that even though we did not compute parity for diagonal 4, we did not require the parity of diagonal 4 to complete the reconstruction of all the missing blocks. This turns out to be true for all pairs of failed disks: we never need to use the parity of the missing diagonal to complete reconstruction. Therefore, we can safely ignore one diagonal during parity construction.

5 Proof of Correctness

Let us formalize the construction of the array. We construct an array of $p + 1$ disks divided into blocks, where p is a prime number greater than 2. We group the blocks at the same position in each device into a stripe. We then take groups of $p - 1$ stripes and, within that group of stripes, assign the blocks to diagonal parity sets such that with disks numbered $i = 0 \ldots p$ and blocks numbered $k = 0 \ldots p - 2$ on each disk, disk block (i, k) belongs to diagonal parity set $(i + k) \bmod p$.

Disk p is a special diagonal parity disk. We construct row parity sets across disks 0 to $p-1$ without involving disk p, so that any one lost block of the first p disks can be reconstructed from row parity. The normal way to ensure this is to store a single row parity block in one of the blocks in each stripe. Without loss of generality, let disk $p-1$ store row parity.

The key observation is that the diagonal parity disk can store diagonal parity for all but one of the p diagonals. Since the array only has $p-1$ rows, we can only store $p-1$ of the p possible diagonal parity blocks in each group of $p-1$ stripes. We could select any of the diagonal parity blocks to leave out, but without loss of generality, we choose to not store parity for diagonal parity set $p-1$, to conform to our numbering scheme.

The roles of all the disks other than the diagonal parity disk are mathematically identical, since they all contribute symmetrically to the diagonal parity disk, and they all contribute to make the row parity sums zero. So, in any stripe any one or more of the non-diagonal parity disks could contain row parity. We only require that we be able to reconstruct any one lost block in a stripe other than the diagonal parity block from row parity without reference to the diagonal parity block.

We start the proof of the correctness of the RDP algorithm with a necessary Lemma.

Lemma 1 *In the sequence of numbers* $\{(p-1+kj) \bmod p, k = 0 \ldots p\}$, *with* p *prime and* $0 < j < p$, *the endpoints are both equal to* $p-1$, *and all numbers* $0 \ldots p-2$ *occur exactly once in the sequence.*

Proof: The first number in the sequence is $p-1$ by definition. The last number in the sequence is $p-1$, since $(p-1+pj) \bmod p = p-1+(pj \bmod p) = p-1$. Thus the lemma is true for the two endpoints. Now consider the subsequence of $p-1$ numbers that begins with $p-1$. All these numbers must have values $0 \le x \le p-1$ after the modulus operation. If there were a repeating number x in the sequence, then it would have to be true that $(x+kj) \bmod p = x$ for some $k < p$. Therefore, $kj \bmod p = 0$ which means that kj is divisible by p. But since p is prime, no multiple of k or j or any of their factors can equal p. Therefore, the first $p-1$ numbers in the sequence beginning with $p-1$ are unique, and all numbers from $0 \ldots p-1$ are represented exactly once. The next number in the sequence is $p-1$. $\diamond$

We now complete the proof of the correctness of RDP.

Theorem 1 *An array constructed according to the formal description of RDP can be reconstructed after the loss of any two of its disks.*

Proof: There are two classes of double failures, those that include the diagonal parity disk, and those that do not.

Those failures that include the diagonal parity disk have only one disk that has failed in the row parity section of the array. This disk can be reconstructed from row parity, since the row parity sets do not involve the diagonal parity disk. Upon completion of the reconstruction of one of the failed disks from row parity, the diagonal parity disk can be reconstructed according to the definition of the diagonal parity sets.

This leaves all failures of any two disks that are not the diagonal parity disk.

From the construction of the array, each disk d intersects all diagonals except diagonal $(d+p-1) \bmod p = (d-1) \bmod p$. Therefore, each disk misses a different diagonal.

For any combination of two failed disks d_1, d_2 with $d_2 = d_1 + j$, the two diagonals that are not intersected by both disks are

$$\begin{aligned} g_1 &= (d_1 + p - 1) \bmod p \\ g_2 &= (d_1 + j + p - 1) \bmod p \end{aligned}$$

Substituting g_1 gives

$$g_2 = (g_1 + j) \bmod p$$

Since each of these diagonals is only missing one member, if we have stored diagonal parity for the diagonal we can reconstruct the

missing element along that diagonal. Since at most one of the diagonals is diagonal $p-1$, then we can reconstruct at least one block on one of the missing disks from diagonal parity as the first step of reconstruction.

For the failed disks d_1, d_2, if we can reconstruct a block from diagonal parity in diagonal parity set x on disk d_1, then we can reconstruct a block on disk d_2 in diagonal parity set $(x+j) \bmod p$, using row parity. Similarly, if we can reconstruct a block x from diagonal parity on disk d_2, then we can reconstruct a block on disk d_1 in diagonal parity set $(x-j) \bmod p$ using row parity.

Consider the pair of diagonals g_1, g_2 that are potentially reconstructable after the failure of disks d_1, d_2. If g_1 is reconstructable, then we can reconstruct all blocks on each diagonal $(g_1 - j) \bmod p, (g_1 - 2j) \bmod p, \ldots, p-1$ using alternating row parity and diagonal parity reconstructions. Similarly, if g_2 is reconstructable, then we can reconstruct all blocks on each diagonal $(g_2 + j) \bmod p, (g_2 + 2j) \bmod p, \ldots, p-1$ using alternating row parity and diagonal parity reconstructions. Since g_1 and g_2 are adjacent points on the sequence for j generated by Lemma 1, then we reach all diagonals $0 \ldots p-1$ during reconstruction.

If either $g_1 = p-1$ or $g_2 = p-1$, then we are only missing one block from the diagonal parity set $p-1$, and that block is reconstructed from row parity at the end of the reconstruction chain beginning with g_2 or g_1 respectively. If both $g_1 \neq p-1$ and $g_2 \neq p-1$, then the reconstruction proceeds from both g_1 and g_2, reaching the two missing blocks on diagonal $p-1$ at the end of each chain. These two blocks are each reconstructed from row parity.

Therefore, all diagonals are reached during reconstruction, and all missing blocks on each diagonal are reconstructed. $\diamond$

We do not need to store or generate the parity of diagonal $p-1$ to complete reconstruction.

6 Performance Analysis

Performance of disk arrays is a function of disk I/O as well as the CPU and memory bandwidth required to construct parity during normal operation and to reconstruct lost data and parity after failures. In this section, we analyse RDP in terms of both its I/O efficiency and its compute efficiency.

Since RDP stores data in the clear, read performance is unaffected by the algorithm, except to the extent that the disk reads and writes associated with data writes interfere with data read traffic. We consider write I/Os for the case where $p-1$ RDP stripes are contained within a single stripe of disk blocks, as described in Section 7. This implementation optimizes write I/O, and preserves the property that any stripe of disk blocks can be written independently of all other stripes. Data writes require writing two parity blocks per stripe. Full stripe writes therefore cost one additional disk I/O compared to full stripe writes in single disk parity arrays. Partial stripe writes can be computed by addition, i.e. recomputing parity on the entire stripe, or subtraction, i.e. computing the delta to the parity blocks from the change in each of the data blocks written to, depending on the number of blocks to be written in the stripe. Writes using the subtraction method are commonly referred to as "small writes". Writing d disk blocks by the subtraction method requires $d+2$ reads and $d+2$ writes. The addition method requires $n-d-2$ reads, and $d+2$ writes to write d disk blocks. If reads and writes are the same cost, then the addition method requires n I/Os, where n is the number of disks in the array, and the subtraction method requires $2d+4$ I/Os. The breakpoint between the addition and subtraction method is at $d = (n-4)/2$. The number of disk I/Os for RDP is minimal for a double failure protection algorithm; writing any one data block requires updating both parity blocks, since each data block must contribute to both parity blocks.

We next determine the computational cost of RDP as the total number of exclusive or (xor) operations needed to construct parity. Each data block contributes to one row par-

ity block. In an array of size $p-1$ rows $\times$ $p+1$ disks, there are $p-1$ data blocks per row, and $p-2$ xor operations are required to reduce those blocks to one parity block. Row parity thus requires $(p-1)(p-2) = p^2-3p+2$ xors. We also compute $p-1$ diagonal parity blocks. Each diagonal contains a total of $p-1$ data or row parity blocks, requiring $p-2$ xors to reduce to one diagonal parity block. Therefore diagonal parity construction requires the same number of xors as row parity construction, $(p-1)(p-2) = p^2-3p+2$. The total number of xors required for construction is $2p^2-6p+4$.

Theorem 2 *For an array of n data disks, a ratio of $2-2/n$ xors per block is the minimum number of xors to provide protection against two failures.*

Proof: Assume that we construct parity in the $n+2$ disk array within groups of r rows. We have a minimum of two parity blocks per row of n data blocks, from the Singleton bound. Each data block must contribute to at least two different parity blocks, one on each parity disk, to ensure that we can recover if the data block and one parity block is lost. Any pair of data blocks that contributes to two different parity blocks provides no additional information since losing both data blocks will make all the parity sets they contribute to ambiguous. Therefore, we need to construct $2r$ parity blocks from equations that in the minimal formulation contain no common pairs of data blocks. Since this allows no common subterms between any equations in the minimal formulation, the minimum number of separately xored input terms required to construct the $2r$ parity blocks is $2nr$. A set of $2r$ equations that reduces $2nr$ terms to $2r$ results using xors requires $2nr-2r$ xors. Therefore, the minimum number of xors per data block to achieve double parity protection is:

$$\frac{2nr-2r}{nr} = 2-\frac{2}{n} \qquad (7)$$

◇

RDP protects $(p-1)^2$ data blocks using $2p^2-6p+4$ xors. Setting $n = p-1$, we get $2n^2-2n$ xors to protect n^2 data blocks, which meets the optimal ratio of $2-2/n$.

Data disks	RDP	EVENODD	Difference
4	6	6.67	11.1%
6	10	10.8	8.0%
8	14	14.86	6.1%
12	22	22.91	4.1%
16	30	30.93	3.1%

Table 1: Per Row XOR Counts for Parity Construction

We can compare RDP to the most computationally efficient previously known algorithm, EVENODD. For an array with n data disks, each with $n-1$ data blocks, EVENODD requires $(n-1)(n-1)$ xors to compute row parity, and $(n-2)n$ xors to compute the parity of the n diagonals. [1] EVENODD then requires a further $n-1$ xors to add the parity of one distinguished diagonal to the parity of each of the other $n-1$ diagonals to complete the calculation of stored diagonal parity. This results in a total of $2n^2-3n$ xors to construct parity in EVENODD for an $n(n-1)$ block array. Therefore, EVENODD requires $(2n^2-3n)/(n^2-n) = 2-1/(n-1)$ xors per block.

The two algorithms both have an asymptotic cost of two xors per block. However, the difference in computational cost is significant for the small values of n typical in disk arrays, as shown in Table 1, ignoring the fact that the two algorithms do not function correctly for the same array sizes.

RDP's computational cost of reconstruction after a failure is also optimal. Reconstruction from any single disk failure requires exactly $(p-1)(p-2) = p^2-3p+2$ xors, since each of the $p-1$ lost row parity sets or diagonal parity sets are of the same size p, and we must reconstruct the lost block in each by xoring the surviving $p-1$ blocks, using $p-2$ xor operations. Again setting $n = p-1$, we are recovering n blocks with n^2-n xors, which is $n-1$ xors per parity block. Since we have already shown that we have the minimum number of xors for construction of an array that double protects parity, and since

[1] This is fewer operations than the result given in the EVENODD paper [5], which we believe overcounts the number of xors required to compute parity along a diagonal.

Data disks	RDP	EVENODD	Difference
4	6	9.67	61.2%
6	10	13.80	83.0%
8	14	17.86	27.6%
12	22	25.91	17.8%
16	30	33.93	13.1%

Table 2: Per Row XOR Counts for Data Reconstruction

all parity sets are the same size, then the cost to repair any one lost disk is the same and is also a minimum. We can't make the individual parity sets any smaller and still protect against double failures, and we are reconstructing each block from exactly one parity set. This is true for any disk we might lose.

Reconstructing from any double failure that includes the diagonal parity disk is exactly the same cost as parity construction, since we first reconstruct the lost data or row parity disk from row parity, then reconstruct the diagonal parity disk. Reconstructing any of the data disks from row parity has the same cost as contructing row parity.

The cost of reconstructing any combination of two data or row parity disks can also be determined. We have to reconstruct exactly $2(p-1)$ blocks. Each parity set is of size $p-1$, so the cost to reconstruct each block is again $p-2$ xors. This gives us exactly the same computational cost as construction, and as the other reconstruction cases: $2p^2-6p+4 = 2n^2-2n$ xors. Again, this is optimal.

Comparing again to EVENODD, using the data reconstruction algorithm described in the EVENODD paper, we see an advantage for RDP, as shown in Table 2.

For the numbers of disks that are typical in disk arrays, the performance of the RDP and EVENODD in construction and reconstruction is significantly different. Both are much lower in compute cost than Reed-Solomon coding [5]. RDP is optimal both in compute efficiency and I/O efficiency, during construction in normal operation and reconstruction after a failure.

7 Algorithm Extensions

Single Stripe Implementation: Selecting p to be one of the primes that meets the condition $p = 2^n + 1$ for some n (e.g. 5, 17, 257), allows us to define diagonal parity sets within a group of 2^n stripes. This allows us to define the block size for RDP purposes to be the usual system block size divided by 2^n. Since the disk block sizes are usually powers of two, we can define a self-contained RDP parity set within a single stripe of blocks. For example, if the system's disk block size is 4kB, we can select $p = 17$, giving us 16 RDP blocks per stripe, with each RDP block containing 256 bytes. This allows us to construct an array using all the existing software and techniques for reading and writing a single stripe, adding one disk to contain diagonal parity.

Multiple Row Parity Groups: RDP requires that we be able to recover a single lost block in a stripe using row parity in any case where that block is not on the diagonal parity disk. In both the RAID-4 and RAID-5 style configurations, it is possible to have more than one row parity set per stripe, each with its own row parity block. This means that the portion of the array that does not include the diagonal disk can use any single disk reconstruction technique, including concatenation of more than one RAID-4 or RAID-5 array, or declustered parity techniques [11]. We define diagonal parity sets across all of these disks, and construct diagonal parity from these sets, regardless of whether the individual blocks stored are parity or data blocks. This allows a cost-performance tradeoff between minimizing the number of parity disks, and making reconstruction from single failures faster while still protecting against double failures in the wider array. In such an array, double failures that affect disks in two different row parity sets can be repaired directly from row parity.

There is one other technique for expanding diagonal parity to cover more than one row parity group. Imagine that we have several RDP arrays, all with the same file system block size, although not necessarily with the same value of p. If we xor all of their diagonal

parity blocks together, we will get a single diagonal parity block. We could store a single diagonal parity disk, which is the combination of the diagonal parity disks of each of the constituent arrays. Storing this is sufficient to allow reconstruction from any double disk loss in the array. Any two failures that occur in two different subarrays can be recovered by local row parity. Any two failures that occur in the same subarray must be recovered by diagonal parity. The diagonal parity block for any subarray can be reconstructed by constructing the diagonal parity for each of the intact subarrays, and subtracting it from the stored merged diagonal parity disk. Once we have reconstructed the needed diagonal parity contents, we use normal RDP reconstruction to rebuild the lost blocks of the subarray that we are reconstructing.

Expandable Arrays: The discussion so far has implied that the number of disks in an array is fixed at $p + 1$ for any selection of p. This is not actually the case. We can underpopulate an RDP array, putting fewer data disks than the maximum allowed in the array for a given value of p. $p + 1$ simply sets the maximum array size for a given value of p. When we underpopulate an array, we are taking advantage of the fact that given fewer than $p - 1$ data disks, we could fill the remainder of the array with unused disks that contain only zeros. Since the zero-filled disks contribute to parity blocks, but do not change the contents of any parity block, we can remove them from the array while still imputing their zero-filled contents in our parity calculations. This allows us to expand the array later by adding a zero-filled disk, and adjusting parity as we later write data to that disk.

By the same reasoning, it is allowable to have disks of different sizes in the array. The diagonal parity disk must be one of the largest disks in the array, and all rows must have at least one row parity block. The contributions of the smaller disks to stripes that do not contain blocks from those disks are counted as zeros.

By selecting $p = 257$, we allow the RDP array to grow to up to 255 data disks. This is a sufficiently large number to accomodate any expected disk array size.

8 Implementation Experience

RDP has been implemented as a new feature of Network Appliance's data storage system software (Data ONTAP) version 6.5. Data ONTAP is a complete software system, including an operating system kernel, networking, storage, file system, file system protocols, and RAID code. The RAID layer manages the final layout of data and parity within RAID groups. A volume consists of one or more RAID groups, and each RAID group is independently recoverable. This section contains some observations we made that improved the implementation of the algorithm.

During parity construction, most of the subblocks of each data disk contribute to both a row parity subblock and a diagonal parity subblock. We also note that the contributions of the subblocks of any data disk to the diagonal parity disk are ordered in sequence. This allows us to perform parity construction in a memory efficient manner on modern microprocessors. We process each data block in one pass, xoring its contents into both the row parity and diagonal parity destination blocks. By properly tuning the code to the microprocessor, it is possible to work on all three blocks in the top level of CPU cache. We work on one data block at a time, incrementally constructing the two target parity blocks, which remain in cache. The latency of memory operations gives us a budget for completing two xor operations per 128 bit field on the Pentium 4. We further optimize by ensuring that the data bytes are xored into both destinations once loaded into processor registers. In our implementation, we had enough instructions available per cache line load to complete both xors and a data integrity checksum calculation on the data in each cache line, without a significant loss of performance. This overlap of cpu execution and memory operations greatly reduced the effective cost of computing the second redundant parity block in RDP.

Another observation is that, having protected the array against double failures, our remaining vulnerability is to triple and higher order failures. Whole-disk/media failures are

corrected as they are encountered, by resorting to using RDP for reconstruction only in those stripes that are missing two blocks. The remainder of the missing disk can be reconstructed using row parity, unless it is the diagonal parity disk.

In the case of whole-disk/whole-disk failures, reconstruction of the first disk to fail typically is already underway when the second disk fails. In our implementation, reconstruction from a single disk failure starts from block number 0 and proceeds sequentially to the last disk block. When the second failure occurs, the first stripes of the array are missing only one block, since we have completed reconstruction of the single failure in some stripes. So, we only need to run RDP reconstruction on the remaining stripes of the array. Stripes with two missing blocks are always reconstructed before those with one missing block, reducing the window of vulnerability to a third disk failure. All combinations of disk failures are handled, including those involving reconstructing disks.

Existing RAID-4 and RAID-5 arrays can be easily upgraded to RDP by constructing the diagonal parity disk, using the same code as is used for reconstructing from single diagonal disk failures. Downgrading from RDP to a single parity protection scheme is as simple as removing the diagonal disk.

9 Measured Performance

Data ONTAP version 6.5 runs on a variety of hardware platforms. The current highest performing platform is the FAS980, which includes two 2.8GHz Intel Pentium 4 CPUs per file server (filer). At any time, up to one full CPU can be running RAID code, including xor calculations for RDP. We ran several performance benchmarks using the implementation of RDP in Data ONTAP 6.5.

The first set of experiments is *xortest*, which is our own synthetic benchmark for testing RAID xor and checksum code. The checksum is a modified Adler checksum that is 64 bits wide, computed on each input block. The input is a stripe that is one block deep by n blocks wide. Blocks are 4kB. The RDP prime is 257, and we divide each 4kB block into 256 sixteen byte subblocks for RDP calculations. The *xortest* experiment is run with cold caches, using random data generated in memory. There is no disk access in the test; it is simply a test of memory and CPU processing speed.

We ran two sets of tests. In the first, we computed parity from the input blocks, and also computed the checksum of each input block and the output parity blocks. In the second, we computed no checksums, only parity. In each set of tests, we computed single parity (RAID-4), double parity (RDP), and also performed RDP reconstruction on two randomly selected missing data blocks. We repeated each experiment five times and took the best results, to eliminate the effects of other activity in the operating system. Generally, the best results are repeatable, with a few bad outliers that represent experiments that were interfered with by other activity affecting the processor and cache.

Figures 2 and 3 present the results of the experiments. Note that all the graphs are very linear, with a small offset due to fixed overhead in the algorithm. In each case, the single parity calculation of RAID-4 is fastest. Table 3 shows the measured calculation rates for the various operations. Note that the RDP reconstruction rate is very close to the RDP construction rate. The difference in timings between the two is due primarily to the completion step in reconstruction, which requires a series of operations on the 16 byte RDP blocks. This step is required regardless of the number of blocks in the stripe. Otherwise, the per block computations during RDP construction and reconstruction are basically the same in our implementation. The reconstruction completion step is accounted for in the overhead per operation, determined as the time taken by a hypothetical calculation on a stripe of zero data blocks. The overhead for RDP reconstruction is significantly higher due to the completion step in both cases (Table 3).

The construction and reconstruction rates are close to those obtained for RAID-4 con-

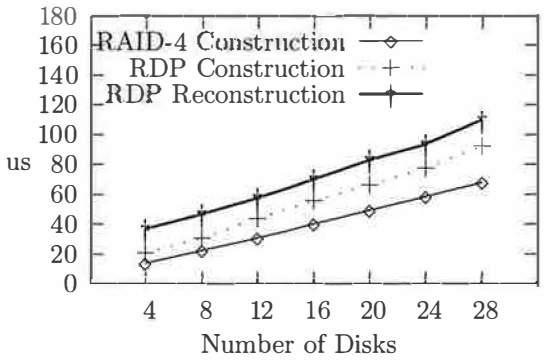


Figure 2: *Xortest* RAID-4 and RDP Performance with Checksum Computations (One 4k block per disk)

		Rate (GB/s)	Overhead (μs)
With Checksum	RAID-4	1.82	4.7
	RDP cons.	1.39	8.3
	RDP recons.	1.37	23.0
Without Checksum	RAID-4	1.90	2.6
	RDP cons.	1.55	4.6
	RDP recons.	1.60	19.8

Table 3: *Xortest* Derived Results

Algorithm	Config. $g \times (d+p)$	Rate (MB/s)
RAID-4	$6 \times (7+1)$	158.3
RDP	$6 \times (7+2)$	149.1
RDP	$4 \times (10+2)$	155.1
RDP	$3 \times (14+2)$	157.2

Table 4: *Aggwrite* Results

struction. (RAID-4 construction and reconstruction are identical computations.) The difference in the rates reflects our inability to completely overlap computation with cache loads and stores to main memory. The theoretical memory bandwidth of the processor is 3.2 GB/s. We are achieving from 43 to 59 percent of this rate, which indicates that we are stalling on cache line loads or are saturating the processor. A calculation of the instruction counts per cache line indicates that we are consuming all of the processing budget available per cache line in the checksum cases.

Aggwrite is a test of the filer's aggregate write performance. The workload is supplied by an array of NFS clients, performing 32kB write operations. Again, these tests are performed using an FAS980 with dual 2.8GHz Intel Pentium 4 processors. The filer runs the entire data path, from network, NFS protocol, file system, RAID and storage. We compared RAID-4 with various configurations of RDP, using 40 or 42 data disks in each case, and measured the achievable write bandwidth. Table 4 gives the aggwrite results.

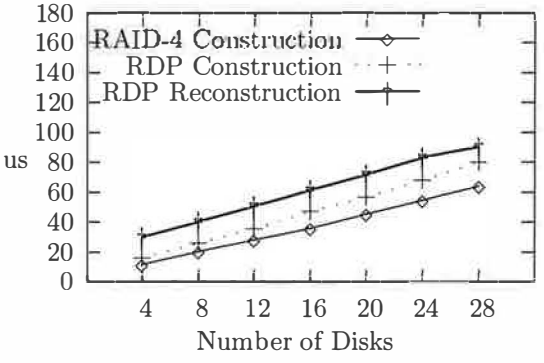


Figure 3: *Xortest* RAID-4 and RDP Performance without Checksum Computations (One 4k block per disk)

The configuration column of Table 4 presents $g \times (d + p)$, where g is the number of separate RAID groups connected to the filer, d is the number of data disks per RAID group, and p is the number of parity disks per RAID group. The WAFL file system uniformly distributes writes across all the data disks. Table 4 indicates that in all cases, RDP performance is within 6 percent of RAID-4. With RDP, we can increase the size of the RAID groups, still realizing an increase in data protection, while achieving comparable write performance. Using RDP RAID groups of 16 disks (14 data and 2 parity) we achieve performance almost equivalent to RAID-4, with the same total number of data and parity disks, and with much improved data protection.

10 Conclusions

RDP is an optimally efficient double disk failure protection algorithm. The combination of a single whole-disk failure with one or more media failures is becoming particularly troublesome as disks get large relative to the expected bit error rate. Utilizing RDP, we can significantly reduce data loss due to all types of double failures. The fact that RDP is optimal in I/O and in computational complexity proved valuable in achieving performance that is very close to our single parity RAID-4 implementation. The simplicity and flexibility of RDP allowed us to implement it within our existing RAID framework. An interesting open problem is whether the algorithm can be extended to cover three or more concurrent failures.

11 Acknowledgements

The authors would like to thank Loellyn Cassell, Ratnesh Gupta, Sharad Gupta, Sanjay Kumar, Kip Macy, Steve Rogridues, Divyesh Shah, Manpreet Singh, Steve Strange, Rajesh Sundaram, Tom Theaker, and Huan Tu for their significant contributions to this project.

References

[1] K. Salem, and H. Garcia-Molina, "Disk striping", *Proceedings of the 2nd Internation Conference on Data Engineering*, pgs.336-342, Feb. 1986.

[2] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)". In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pgs. 109-116, 1988.

[3] W. Burkhard, and J. Menon, "Disk array storage system reliability". *Proceedings of the International Symposium on Fault-tolerant Computing*, pgs.432-441, 1993.

[4] Qin Xin, E. Miller, T. Schwarz, D. Long, S. Brandt, W. Litwin, "Reliability mechanisms for very large storage systems", *20th IEEE/11th NASA Boddard Conference on Mass Storage Systems and Technologies*, San Diego, CA, pgs. 146-156, Apr. 2003.

[5] M. Blaum, J. Brady, J. Bruck, and J. Menon. "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures". In *Proc. of the Annual International Symposium on Computer Architecture*, pgs. 245-254, 1994.

[6] F. J. MacWilliams and J. J. A. Sloane. *The Theory of Error-Corrrecting Codes*, North-Holland, 1977.

[7] G. A. Alvarez, W. A. Burkhard, and F. Christian, "Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering" *Proceedings of the 24th Annual Symposium on Computer Architecture* pgs. 62-72, 1997.

[8] C. I. Park, "Efficient placement of parity and data to tolerate two disk failures in disk array systems". *IEEE Transactions on Parallel and Distributed Systems*, Nov. 1995.

[9] R. Muntz, and J. Lui, "Performance analysis of disk arrays under failure." *Proceedings of the 16th VLDB Conference* pgs.162-173, Brisbane, June 1990.

[10] D. Anderson, J. Dykes, and E. Riedel, "More than an interface - SCSI vs. ATA". *2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA. pgs. 245-257, March, 2003.

[11] M. Holland, and G. Gibson, "Parity deculstering for continuous operation on redundant disk arrays", *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. pgs. 23-25, 1992.

[12] A. Papoulis, *Probability, Random Variables, and Stochastic Processes, Second Edition*, McGraw-Hill, New York, 1984.

Improving Storage System Availability with D-GRAID

Muthian Sivathanu, Vijayan Prabhakaran,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison
{*muthian, vijayan, dusseau, remzi*}*@cs.wisc.edu*

Abstract

We present the design, implementation, and evaluation of D-GRAID, a gracefully-degrading and quickly-recovering RAID storage array. D-GRAID ensures that most files within the file system remain available even when an unexpectedly high number of faults occur. D-GRAID also recovers from failures quickly, restoring only live file system data to a hot spare. Both graceful degradation and live-block recovery are implemented in a prototype SCSI-based storage system underneath unmodified file systems, demonstrating that powerful "file-system like" functionality can be implemented behind a narrow block-based interface.

1 Introduction

> "If a tree falls in the forest and no one hears it, does it make a sound?" *George Berkeley*

Storage systems comprised of multiple disks are the backbone of modern computing centers, and when the storage system is down, the entire center can grind to a halt. Downtime is clearly expensive; for example, in the on-line business world, millions of dollars per hour are lost when systems are not available [26, 34].

Storage system *availability* is formally defined as the mean time between failure (MTBF) divided by the sum of the MTBF and the mean time to recovery (MTTR): $\frac{MTBF}{MTBF+MTTR}$ [17]. Hence, to improve availability, one can either increase the MTBF or decrease the MTTR. Not surprisingly, researchers have studied both of these components of availability.

To increase the time between failures of a large storage array, data redundancy techniques can be applied [4, 6, 8, 18, 22, 31, 32, 33, 43, 47]. By keeping multiple copies of blocks, or through more sophisticated redundancy schemes such as parity-encoding, storage systems can tolerate a (small) fixed number of faults. To decrease the time to recovery, "hot spares" can be employed [21, 29, 32, 36]; when a failure occurs, a spare disk is activated and filled with reconstructed data, returning the system to normal operating mode relatively quickly.

However, the narrow interface between file systems and storage [13] has curtailed opportunities for improving MTBF and MTTR. In a RAID-5 storage array, if one disk too many fails before another is repaired, the entire array is corrupted. This "availability cliff" is a result of the storage system laying out blocks oblivious of their semantic importance or relationship; most files become corrupted or inaccessible after just one extra disk failure. Until a time-consuming restore from backup, the entire array remains unavailable, although most disks are still operational. Further, because the storage array has no information on which blocks are live in the file system, the recovery process must restore all blocks in the disk. This unnecessary work slows recovery and reduces availability.

An ideal storage array fails gracefully: if $\frac{1}{N}$th of the disks of the system are down, at most $\frac{1}{N}$th of the data is unavailable. An ideal array also recovers more intelligently, restoring only live data. In effect, more "important" data is less likely to disappear under failure, and such data is restored earlier during recovery. This strategy for data availability stems from Berkeley's observation about falling trees: if a file isn't available, and no process tries to access it before it is recovered, is there truly a failure?

To explore these concepts and provide a storage array with more graceful failure semantics, we present the design, implementation, and evaluation of D-GRAID, a RAID system that Degrades Gracefully (and recovers quickly). D-GRAID exploits semantic intelligence [44] within the disk array to place file system structures across the disks in a fault-contained manner, analogous to the fault containment techniques found in the Hive operating system [7] and in some distributed file systems [24, 42]. Thus, when an unexpected "double" failure occurs [17], D-GRAID continues operation, serving those files that can still be accessed. D-GRAID also utilizes semantic knowledge during recovery; specifically, only blocks that the file system considers live are restored onto a hot spare. Both aspects of D-GRAID combine to improve the effective availability of the storage array. Note that D-GRAID techniques are complementary to existing redun-

dancy schemes; thus, if a storage administrator configures a D-GRAID array to utilize RAID Level 5, any single disk can fail without data loss, and additional failures lead to a proportional fraction of unavailable data.

In this paper, we present a prototype implementation of D-GRAID, which we refer to as *Alexander*. Alexander is an example of a semantically-smart disk system [44]. Built underneath a narrow block-based SCSI storage interface, such a disk system understands file system data structures, including the super block, allocation bitmaps, inodes, directories, and other important structures; this knowledge is central to implementing graceful degradation and quick recovery. Because of their intricate understanding of file system structures and operations, semantically-smart arrays are tailored to particular file systems; Alexander currently functions underneath unmodified Linux ext2 and VFAT file systems.

We make three important contributions to semantic disk technology. First, we deepen the understanding of how to build semantically-smart disk systems that operate correctly even with imperfect file system knowledge. Second, we demonstrate that such technology can be applied underneath widely varying file systems. Third, we demonstrate that semantic knowledge allows a RAID system to apply different redundancy techniques based on the type of data, thereby improving availability.

There are two key aspects to the Alexander implementation of graceful degradation. The first is *selective meta-data replication*, in which Alexander replicates naming and system meta-data structures of the file system to a high degree while using standard redundancy techniques for data. Thus, with a small amount of overhead, excess failures do not render the entire array unavailable. Instead, the entire directory hierarchy can still be traversed, and only some fraction of files will be missing, proportional to the number of missing disks. The second is a *fault-isolated data placement* strategy. To ensure that semantically meaningful data units are available under failure, Alexander places semantically-related blocks (*e.g.*, the blocks of a file) within the storage array's unit of fault-containment (*e.g.*, a disk). By observing the natural failure boundaries found within an array, failures make semantically-related groups of blocks unavailable, leaving the rest of the file system intact.

Unfortunately, fault-isolated data placement improves availability at a cost; related blocks are no longer striped across the drives, reducing the natural benefits of parallelism found within most RAID techniques [15]. To remedy this, Alexander also implements *access-driven diffusion* to improve throughput to frequently-accessed files, by spreading a copy of the blocks of "hot" files across the drives of the system. Alexander monitors access to data to determine which files to replicate in this fashion, and finds space for those replicas either in a pre-configured *performance reserve* or opportunistically in the unused portions of the storage system.

We evaluate the availability improvements possible with D-GRAID through trace analysis and simulation, and find that D-GRAID does an excellent job of masking an arbitrary number of failures from most processes by enabling continued access to "important" data. We then evaluate our prototype Alexander under microbenchmarks and trace-driven workloads. We find that the construction of D-GRAID is feasible; even with imperfect semantic knowledge, powerful functionality can be implemented within a block-based storage array. We also find that the run-time overheads of D-GRAID are small, but that the CPU costs as compared to a standard array are high. We show that access-driven diffusion is crucial for performance, and that live-block recovery is effective when disks are under-utilized. The combination of replication, data placement, and recovery techniques results in a storage system that improves availability while maintaining a high level of performance.

The rest of this paper is structured as follows. In Section 2, we present extended motivation, and in Section 3, we discuss the design principles of D-GRAID. In Section 4, we present trace analysis and simulations, and discuss semantic knowledge in Section 5. In Section 6, we present our prototype implementation. We evaluate our prototype in Section 7, discuss alternative methods to implementing D-GRAID and the commercial feasibility of a semantic disk based approach in Section 8. In Section 9, we present related work and conclude in Section 10.

2 Extended Motivation

The Case for Graceful Degradation: RAID redundancy techniques typically export a simple failure model. If D or fewer disks fail, the RAID continues to operate correctly, but perhaps with degraded performance. If more than D disks fail, the RAID is entirely unavailable until the problem is corrected, perhaps via a restore from tape. In most RAID schemes, D is small (often 1); thus even when most disks are working, users may observe a "failed" disk system.

With graceful degradation, a RAID system can absolutely tolerate some fixed number of faults (as before), and excess failures are not catastrophic; most of the data (an amount proportional to the number of disks still available in the system) continues to be available, thus allowing access to that data while the other "failed" data is restored. It does not matter to users or applications whether the entire contents of the volume are present; rather, what matters is whether a particular set of files are available.

One question is whether it is realistic to expect a catastrophic failure scenario within a RAID system. For example, in a RAID-5 system, given the high MTBF's reported by disk manufacturers, one might believe that a second

disk failure is highly unlikely to occur before the first failed disk is repaired. However, multiple disk failures do occur, for two primary reasons. First, correlated faults are more common in systems than expected [19]. If the RAID has not been carefully designed in an orthogonal manner, a single controller fault or other component error can render a fair number of disks unavailable [8]; such redundant designs are expensive, and therefore may only be found in higher end storage arrays. Second, Gray points out that system administration is the main source of failure in systems [17]. A large percentage of human failures occur during maintenance, where "the maintenance person typed the wrong command or unplugged the wrong module, thereby introducing a double failure" (page 6) [17].

Other evidence also suggests that multiple failures can occur. For example, IBM's ServeRAID array controller product includes directions on how to attempt data recovery when multiple disk failures occur within a RAID-5 storage array [23]. Within our own organization, data is stored on file servers under RAID-5. In one of our servers, a single disk failed, but the indicator that should have informed administrators of the problem did not do so. The problem was only discovered when a second disk in the array failed; full restore from backup ran for days. In this scenario, graceful degradation would have enabled access to a large fraction of user data during the long restore.

One might think that the best approach to dealing with multiple failures would be to employ a higher level of redundancy [2, 6], thus enabling the storage array to tolerate a greater number of failures without loss of data. However, these techniques are often expensive (*e.g.*, three-way data mirroring) or bandwidth-intensive (*e.g.*, more than 6 I/Os per write in a P+Q redundant store). Graceful degradation is complementary to such techniques. Thus, storage administrators could choose the level of redundancy they believe necessary for common case faults; graceful degradation is enacted when a "worse than expected" fault occurs, mitigating its ill effect.

Need for Semantically-Smart Storage: Implementing new functionality in a semantically-smart disk system has the key benefit of enabling wide-scale deployment underneath an unmodified SCSI interface without any OS modification, thus working smoothly with existing file systems and software base. Although there is some desire to evolve the interface between file systems and storage [16], the reality is that current interfaces will likely survive much longer than anticipated. As Bill Joy once said, "systems may come and go, but protocols live forever". A new mechanism like D-GRAID is more likely to be deployed if it is non-intrusive on existing infrastructure; semantic disks ensure just that.

3 Design: D-GRAID Expectations

In this section, we discuss the design of D-GRAID. We present background information on file systems, the data layout strategy required to enable graceful degradation, the important design issues that arise due to the new layout, and the process of fast recovery.

3.1 File System Background

Semantic knowledge is system specific; therefore, we discuss D-GRAID design and implementation for two widely differing file systems: Linux ext2 [45] and Microsoft VFAT [30] file system. Inclusion of VFAT represents a significant contribution compared to previous research, which operated solely underneath UNIX file systems.

The ext2 file system is an intellectual descendant of the Berkeley Fast File System (FFS) [28]. The disk is split into a set of *block groups*, akin to cylinder groups in FFS, each of which contains bitmaps to track inode and data block allocation, inode blocks, and data blocks. Most information about a file, including size and block pointers, are found in the file's inode.

The VFAT file system descends from the world of PC operating systems. In this paper, we consider the Linux VFAT implementation of FAT-32, although our work is general and applies to other variants. VFAT operations are centered around the eponymous file allocation table, which contains an entry for each allocatable block in the file system. These entries are used to locate the blocks of a file, in a linked-list fashion, *e.g.*, if a file's first block is at address b, one can look in entry b of the FAT to find the next block of the file, and so forth. An entry can also hold an end-of-file marker or a setting that indicates the block is free. Unlike UNIX file systems, where most information about a file is found in its inode, a VFAT file system spreads this information across the FAT itself and the directory entries; the FAT is used to track which blocks belong to the file, whereas the directory entry contains information like size, permission, and type information.

3.2 Graceful Degradation

To ensure partial availability of data under multiple failures in a RAID array, D-GRAID employs two main techniques. The first is a *fault-isolated data placement* strategy, in which D-GRAID places each "semantically-related set of blocks" within a "unit of fault containment" found within the storage array. For simplicity of discussion, we assume that a file is a semantically-related set of blocks, and that a single disk is the unit of fault containment. We will generalize the former below, and the latter is easily generalized if there are other failure boundaries that should be observed (*e.g.*, SCSI chains). We refer to the physical disk to which a file belongs as the *home site* for the file. When a particular disk fails, fault-isolated data placement ensures that only files that have that disk

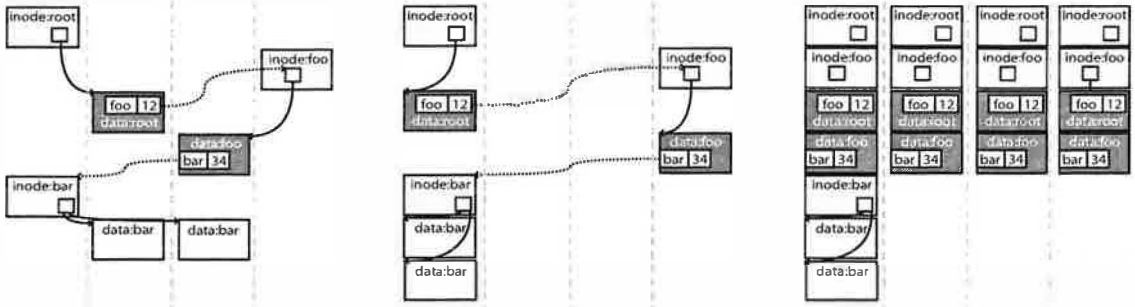


Figure 1: A Comparison of Layout Schemes. *These figures depict different layouts of a file "/foo/bar" in a* UNIX *file system starting at the root inode and following down the directory tree to the file data. Each vertical column represents a disk. For simplicity, the example assumes no data redundancy for user file data. On the left is a typical file system layout on a non-D-GRAID disk system; because blocks (and therefore pointers) are spread throughout the file system, any single fault will render the blocks of the file "bar" inaccessible. In the middle is a fault-isolated data placement of files and directories. In this scenario, if one can access the inode of a file, one can access its data (indirect pointer blocks would also be constrained within the same disk). Finally, on the right is an example of selective meta-data replication. By replicating directory inodes and directory blocks, D-GRAID can guarantee that users can get to all files that are available. Some of the requisite pointers have been removed from the rightmost figure for simplicity. Color codes are white for user data, light shaded for inodes, and dark shaded for directory data.*

as their home site become unavailable, while other files remain accessible as whole files.

The second technique is *selective meta-data replication*, in which D-GRAID replicates naming and system meta-data structures of the file system to a high degree, *e.g.*, directory inodes and directory data in a UNIX file system. D-GRAID thus ensures that all live data is reachable and not orphaned due to failure. The entire directory hierarchy remains traversable, and the fraction of missing user data is proportional to the number of failed disks.

Thus, D-GRAID lays out logical file system blocks in such a way that the availability of a single file depends on as few disks as possible. In a traditional RAID array, this dependence set is normally the entire set of disks in the group, thereby leading to entire file system unavailability under an unexpected failure. A UNIX-centric example of typical layout, fault-isolated data placement, and selective meta-data replication is depicted in Figure 1. Note that for the techniques in D-GRAID to work, a meaningful subset of the file system must be laid out within a single D-GRAID array. For example, if the file system is striped across multiple D-GRAID arrays, no single array will have a meaningful view of the file system. In such a scenario, D-GRAID can be run at the logical volume manager level, viewing each of the arrays as a single disk; the same techniques remain relevant.

Because D-GRAID treats each file system block type differently, the traditional RAID taxonomy is no longer adequate in describing how D-GRAID behaves. Instead, a finer-grained notion of a RAID level is required, as D-GRAID may employ different redundancy techniques for different types of data. For example, D-GRAID commonly employs n-way mirroring for naming and system meta-data, whereas it uses standard redundancy techniques, such as mirroring or parity encoding (*e.g.*, RAID-5), for user data. Note that n, a value under administrative control, determines the number of failures under which D-GRAID will degrade gracefully. In Section 4, we will explore how data availability degrades under varying levels of namespace replication.

3.3 Design Considerations

The layout and replication techniques required to enable graceful degradation introduce a host of design issues. We highlight the major challenges that arise.

Semantically-related blocks: With fault-isolated data placement, D-GRAID places a logical unit of file system data (*e.g.*, a file) within a fault-isolated container (*e.g.*, a disk). Which blocks D-GRAID considers "related" thus determines which data remains available under failure. The most basic approach is *file-based* grouping, in which a single file (including its data blocks, inode, and indirect pointers) is treated as the logical unit of data; however, with this technique a user may find that some files in a directory are unavailable while others are not, which may cause frustration and confusion. Other groupings preserve more meaningful portions of the file system volume under failure. With *directory-based* grouping, D-GRAID ensures that the files of a directory are all placed within the same unit of fault containment. Less automated options are also possible, allowing users to specify arbitrary semantic groupings which D-GRAID then treats as a unit.

Load balance: With fault-isolated placement, instead of placing blocks of a file across many disks, the blocks are isolated within a single home site. Isolated placement improves availability but introduces the problem of load balancing, which has both space and time components.

In terms of space, the total utilized space in each disk should be maintained at roughly the same level, so that when a fraction of disks fail, roughly the same fraction of data becomes unavailable. Such balancing can be addressed in the foreground (*i.e.*, when data is first allocated), the background (*i.e.*, with migration), or both. Files (or directories) larger than the amount of free space in a single disk can be handled either with a potentially

expensive reorganization or by reserving large extents of free space on a subset of drives. Files that are larger than a single disk must be split across disks.

More pressing are the performance problems introduced by fault-isolated data placement. Previous work indicates that striping of data across disks is better for performance even compared to sophisticated file placement algorithms [15, 48]. Thus, D-GRAID makes additional copies of user data that are spread across the drives of the system, a process which we call *access-driven diffusion*. Whereas standard D-GRAID data placement is optimized for availability, access-driven diffusion increases performance for those files that are frequently accessed. Not surprisingly, access-driven diffusion introduces policy decisions into D-GRAID, including where to place replicas that are made for performance, which files to replicate, and when to create the replicas.

Meta-data replication level: The degree of meta-data replication within D-GRAID determines how resilient it is to excessive failures. Thus, a high degree of replication is desirable. Unfortunately, meta-data replication comes with costs, both in terms of space and time. For space overheads, the trade-offs are obvious: more replicas imply more resiliency. One difference between traditional RAID and D-GRAID is that the amount of space needed for replication of naming and system meta-data is dependent on usage, *i.e.*, a volume with more directories induces a greater amount of overhead. For time overheads, a higher degree of replication implies lowered write performance for naming and system meta-data operations. However, others have observed that there is a lack of update activity at higher levels in the directory tree [35], and lazy update propagation can be employed to reduce costs [43].

3.4 Fast Recovery

Because the main design goal of D-GRAID is to ensure higher availability, fast recovery from failure is also critical. The most straightforward optimization available with D-GRAID is to recover only "live" file system data. Assume we are restoring data from a live mirror onto a hot spare; in the straightforward approach, D-GRAID simply scans the source disk for live blocks, examining appropriate file system structures to determine which blocks to restore. This process is readily generalized to more complex redundancy encodings. D-GRAID can potentially prioritize recovery in a number of ways, *e.g.*, by restoring certain "important" files first, where importance could be domain specific (*e.g.*, files in `/etc`) or indicated by users in a manner similar to the hoarding database in Coda [27].

4 Exploring Graceful Degradation

In this section, we use simulation and trace analysis to evaluate the potential effectiveness of graceful degradation and the impact of different semantic grouping techniques. We first quantify the space overheads of D-GRAID. Then we demonstrate the ability of D-GRAID to provide continued access to a proportional fraction of meaningful data after arbitrary number of failures. More importantly, we then demonstrate how D-GRAID can hide failures from users by replicating "important" data. The simulations use file system traces collected from HP Labs [38], and cover 10 days of activity; there are 250 GB of data spread across 18 logical volumes.

	Level of Replication		
	1-way	4-way	16-way
$ext2_{1KB}$	0.15%	0.60%	2.41%
$ext2_{4KB}$	0.43%	1.71%	6.84%
$VFAT_{1KB}$	0.52%	2.07%	8.29%
$VFAT_{4KB}$	0.50%	2.01%	8.03%

Table 1: **Space Overhead of Selective Meta-data Replication.** *The table shows the space overheads of selective meta-data replication as a percentage of total user data, and as the level of naming and system meta-data replication increases. In the leftmost column, the percentage space overhead without any meta-data replication is shown. The next two columns depict the costs of modest (4-way) and paranoid (16-way) schemes. Each row shows the overhead for a particular file system, either ext2 or VFAT, with block size set to 1 KB or 4 KB.*

4.1 Space Overheads

We first examine the space overheads due to selective meta-data replication that are typical with D-GRAID-style redundancy. We calculate the cost of selective meta-data replication as a percentage overhead, measured across all volumes of the HP trace data. We calculate the highest selective meta-data replication overhead percentage possible by assuming no replication of user data; if user data is mirrored, the overheads are cut in half.

Table 1 shows that selective meta-data replication induces only a mild space overhead even under high levels of meta-data redundancy for both the Linux ext2 and VFAT file systems. Even with 16-way redundancy of meta-data, only a space overhead of 8% is incurred in the worst case (VFAT with 1 KB blocks). With increasing block size, while ext2 uses more space (due to internal fragmentation with larger directory blocks), the overheads actually decrease with VFAT. This phenomenon is due to the structure of VFAT; for a fixed-sized file system, as block size grows, the file allocation table itself shrinks, although the blocks that contain directory data grow.

4.2 Static Availability

We next examine how D-GRAID availability degrades under failure with two different semantic grouping strategies. The first strategy is file-based grouping, which keeps the information associated with a single file within a failure boundary (*i.e.*, a disk); the second is directory-based grouping, which allocates files of a directory together. For this analysis, we place the entire 250 GB of files and directories from the HP trace onto a simulated 32-disk sys-

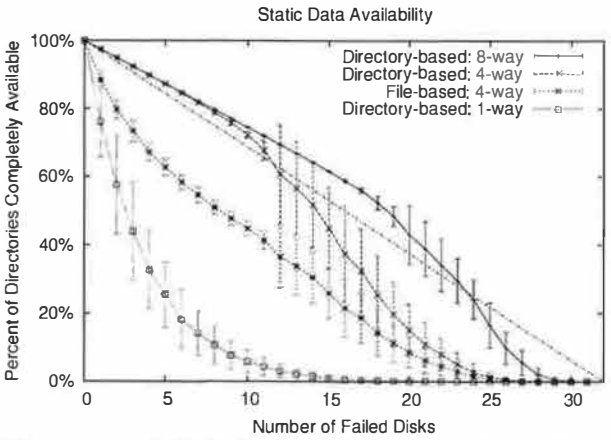


Figure 2: Static Data Availability. *The percent of entire directories available is shown under increasing disk failures. The simulated system consists of 32 disks, and is loaded with the 250 GB from the HP trace. Two different strategies for semantic grouping are shown: file-based and directory-based. Each line varies the level of replication of namespace meta-data. Each point shows average and deviation across 30 trials, where each trial randomly varies which disks fail.*

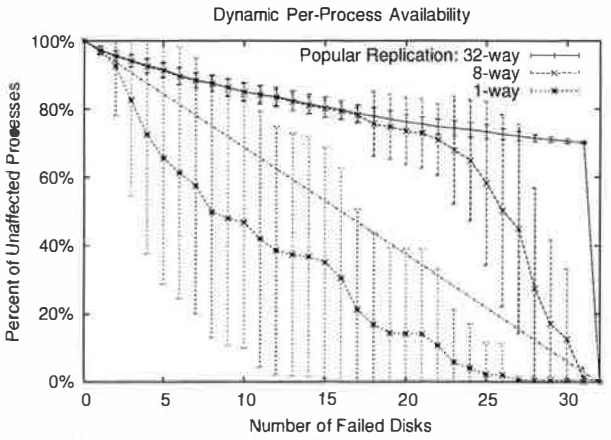


Figure 3: Dynamic Data Availability. *The figure plots the percent of processes that run unaffected under disk failure from one busy hour from the HP trace. The degree of namespace replication is set aggressively to 32. Each line varies the amount of replication for "popular" directories; 1-way implies that those directories are not replicated, whereas 8-way and 32-way show what happens with a modest and extreme amount of replication. Means and deviations of 30 trials are shown.*

tem, remove simulated disks, and measure the percentage of whole directories that are available. We assume no user data redundancy (*i.e.*, D-GRAID Level 0).

Figure 2 shows the percent of directories available, where a directory is available if all of its files are accessible (although subdirectories and their files may not be). From the figure, we observe that graceful degradation works quite well, with the amount of available data proportional to the number of working disks, in contrast to a traditional RAID where a few disk crashes would lead to complete data unavailability. In fact, availability sometimes degrades slightly less than expected from a strict linear fall-off; this is due to a slight imbalance in data placement across disks and within directories. Further, even a modest level of namespace replication (*e.g.*, 4-way) leads to very good data availability under failure. We also conclude that with file-based grouping, some files in a directory are likely to "disappear" under failure, leading to user dissatisfaction.

4.3 Dynamic Availability

Finally, by simulating dynamic availability, we examine how often users or applications will be oblivious that D-GRAID is operating in degraded mode. Specifically, we run a portion of the HP trace through a simulator with some number of failed disks, and record what percent of processes observed no I/O failure during the run. Through this experiment, we find that namespace replication is not enough; certain files, that are needed by most processes, must be replicated as well.

In this experiment, we set the degree of namespace replication to 32 (full replication), and vary the level of replication of the contents of popular directories, *i.e.*, `/usr/bin`, `/bin`, `/lib` and a few others. Figure 3 shows that without replicating the contents of those directories, the percent of processes that run without ill-effect is lower than expected from our results in Figure 2. However, when those few directories are replicated, the percentage of processes that run to completion under disk failure is much better than expected. The reason for this is clear: a substantial number of processes (*e.g.*, `who`, `ps`, etc.) only require that their executable and a few other libraries are available to run correctly. With popular directory replication, excellent availability under failure is possible. Fortunately, almost all of the popular files are in "read only" directories; thus, wide-scale replication will not raise write performance or consistency issues. Also, the space overhead due to popular directory replication is minimal for a reasonably sized file system; for this trace, such directories account for about 143 MB, less than 0.1% of the total file system size.

5 Semantic Knowledge

We now move towards the construction of a D-GRAID prototype underneath a block-based SCSI-like interface. The enabling technology underlying D-GRAID is semantic knowledge [44]. Understanding how the file system above utilizes the disk enables D-GRAID to implement both graceful degradation under failure and quick recovery. The exact details of acquiring semantic knowledge within a disk or RAID system have been described elsewhere [44]; here we just assume that a basic understanding of file system layout and structures is available within the storage system. Specifically, we assume that D-GRAID has static knowledge of file system layout, including which regions on disk are used for which block types and the contents of specific block types, *e.g.*, the fields of an inode.

5.1 File System Behaviors

In this paper, we extend understanding of semantically-smart disks by presenting techniques to handle more general file system behaviors. Previous work required the file system to be mounted synchronously when implementing complex functionality within the disk; we relax that requirement. We now describe our assumptions for general file system behavior; we believe that many, if not all, modern file systems adhere to these behavioral guidelines.

First, blocks in a file system can be dynamically typed, *i.e.*, the file system can locate different types of blocks at the same physical location on disk over the lifetime of the file system. For example, in a UNIX file system, a block in the data region can be a user-data block, an indirect-pointer block or a directory-data block. Second, a file system can delay updates to disk; delayed writes at the file system facilitate batching of small writes in memory and suppressing of writes to files that are subsequently deleted. Third, as a consequence of delayed writes, the order in which the file system actually writes data to disk can be arbitrary. Although certain file systems order writes carefully [14], to remain general, we do not make any such assumptions on the ordering. Note that our assumptions are made for practical reasons: the Linux ext2 file system exhibits all the aforementioned behaviors.

5.2 Accuracy of Information

Our assumptions about general file system behavior imply that the storage system cannot accurately classify the type of each block. Block classification is straightforward when the type of the block depends upon its location on disk. For example, in the Berkeley Fast File System (FFS) [28], the regions of disk that store inodes are fixed at file system creation; thus, any traffic to those regions is known to contain inodes.

However, type information is sometimes spread across multiple blocks. For example, a block filled with indirect pointers can only be identified as such by observing the corresponding inode, specifically that the inode's indirect pointer field contains the address of the given indirect block. More formally, to identify an indirect block B, the semantic disk must look for the inode that has block B in its indirect pointer field. Thus, when the relevant inode block I_B is written to disk, the disk infers that B is an indirect block, and when it later observes block B written, it uses this information to classify and treat the block as an indirect block. However, due to the delayed write and reordering behavior of the file system, it is possible that in the time between the disk writes of I_B and B, block B was freed from the original inode and was reallocated to another inode with a different type, *i.e.*, as a normal data block. The disk does not know this since the operations took place in memory and were not reflected to disk. Thus, the inference made by the semantic disk on the block type could be wrong due to the inherent staleness of the information tracked. Implementing a correct system despite potentially inaccurate inferences is one of the challenges we address in this paper.

6 Implementation: Making D-GRAID

We now discuss the prototype implementation of D-GRAID known as Alexander. Alexander uses fault-isolated data placement and selective meta-data replication to provide graceful degradation under failure, and employs access-driven diffusion to correct the performance problems introduced by availability-oriented layout. Currently, Alexander replicates namespace and system meta-data to an administrator-controlled value (*e.g.*, 4 or 8), and stores user data in either a RAID-0 or RAID-1 manner; we refer to those systems as D-GRAID Levels 0 and 1, respectively. We are currently pursuing a D-GRAID Level 5 implementation, which uses log-structuring [39] to avoid the small-write problem that is exacerbated by fault-isolated data placement.

In this section, we present the implementation of graceful degradation and live-block recovery, with most of the complexity (and hence discussion) centered around graceful degradation. For simplicity of exposition, we focus on the construction of Alexander underneath the Linux ext2 file system. At the end of the section, we discuss differences in our implementation underneath VFAT.

6.1 Graceful Degradation

We now present an overview of the basic operation of graceful degradation within Alexander.

6.1.1 The Indirection Map

Similar to any other SCSI-based RAID system, Alexander presents host systems with a linear logical block address space. Internally, Alexander must place blocks so as to facilitate graceful degradation. Thus, to control placement, Alexander introduces a transparent level of indirection between the logical array used by the file system and physical placement onto the disks via the *indirection map (imap)*; similar structures have been used by others [12, 46, 47]. Unlike most of these other systems, this imap only maps every *live* logical file system block to its replica list, *i.e.*, all its physical locations. All *unmapped* blocks are considered free and are candidates for use by D-GRAID.

6.1.2 Reads

Handling block read requests at the D-GRAID level is straightforward. Given the logical address of the block, Alexander looks in the imap to find the replica list and issues the read request to one of its replicas. The choice of which replica to read from can be based on various criteria [47]; currently Alexander uses a randomized selection.

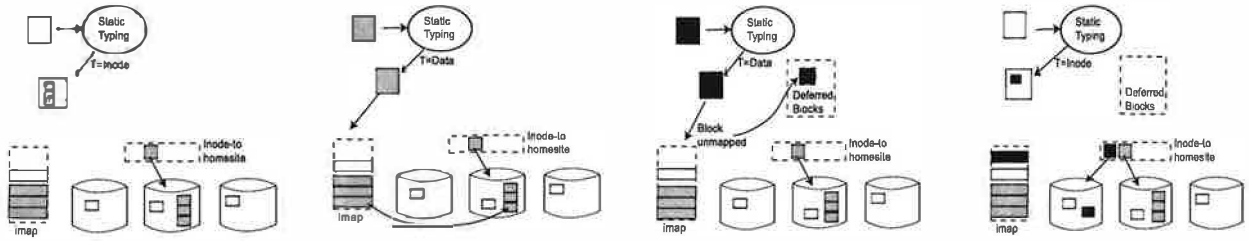


Figure 4: **Anatomy of a Write** *This figure depicts the control flow during a sequence of* `write` *operations in Alexander. In the first figure, an inode block is written; Alexander observes the contents of the inode block and identifies the newly added inode. It then selects a home site for the inode and creates physical mappings for the blocks of the inode, in that home site. Also, the inode block is aggressively replicated. In the next figure, Alexander observes a write to a data block from the same inode; since it is already mapped, the write goes directly to the physical block. In the third figure, Alexander gets a write to an unmapped data block; it therefore defers the block, and when Alexander finally observes the corresponding inode (in the fourth figure), it creates the relevant mappings, observes that one of its blocks is deferred, and therefore issues the deferred write to the relevant home site.*

6.1.3 Writes

In contrast to reads, write requests are more complex to handle. Exactly how Alexander handles the write request depends on the *type* of the block that is written. Figure 4 depicts some common cases.

If the block is a static meta-data block (*e.g.*, an inode or a bitmap block) that is as of yet unmapped, Alexander allocates a physical block in each of the disks where a replica should reside, and writes to all of the copies. Note that Alexander can easily detect static block types such as inode and bitmap blocks underneath many UNIX file systems simply by observing the logical block address.

When an inode block is written, D-GRAID scans the block for newly added inodes; to understand which inodes are new, D-GRAID compares the newly written block with its old copy, a process referred to as block differencing. For every new inode, D-GRAID selects a home site to lay out blocks belonging to the inode, and records it in the *inode-to-homesite* hashtable. This selection of home site is done to balance space allocation across physical disks. Currently, D-GRAID uses a greedy approach; it selects the home site with the least disk space utilization.

If the write is to an unmapped block in the data region (*i.e.*, a data block, an indirect block, or a directory block), the allocation cannot be done until D-GRAID knows which file the block belongs to, and thus, its actual home site. In such a case, D-GRAID places the block in a *deferred block list* and does not write it to disk until it learns which file the block is associated with. Since a crash before the inode write would make the block inaccessible by the file system anyway, the in-memory deferred block list is not a reliability concern.

D-GRAID also looks for newly added block pointers when an inode (or indirect) block is written. If the newly added block pointer refers to an unmapped block, D-GRAID adds a new entry in the imap, mapping the logical block to a physical block in the home site assigned to the corresponding inode. If any newly added pointer refers to a block in the deferred list, D-GRAID removes the block from the deferred list and issues the write to the appropriate physical block(s). Thus, writes are deferred only for blocks that are written *before* the corresponding owner inode blocks. If the inode is written first, subsequent data writes will be already mapped and sent to disk directly.

Another block type of interest that D-GRAID looks for is the data bitmap block. Whenever a data bitmap block is written, D-GRAID scans through it looking for newly freed data blocks. For every such freed block, D-GRAID removes the logical-to-physical mapping if one exists and frees the corresponding physical blocks. Further, if a block that is currently in the deferred list is freed, the block is removed from the deferred list and the write is suppressed; thus, data blocks that are written by the file system but deleted before their corresponding inode is written to disk do not generate extra disk traffic, similar to optimizations found in many file systems [39]. Removing such blocks from the deferred list is important because in the case of freed blocks, Alexander may never observe an owning inode. Thus, every deferred block stays in the deferred list for a bounded amount of time, until either an inode owning the block is written, or a bitmap block indicating deletion of the block is written. The exact duration depends on the delayed write interval of the file system.

6.1.4 Block Reuse

We now discuss a few of the more intricate issues involved with implementing graceful degradation. The first such issue is block reuse. As existing files are deleted or truncated and new files are created, blocks that were once part of one file may be reallocated to some other file. Since D-GRAID needs to place blocks onto the correct home site, this reuse of blocks needs to be detected and acted upon. D-GRAID handles block reuse in the following manner: whenever an inode block or an indirect block is written, D-GRAID examines each valid block pointer to see if its physical block mapping matches the home site allocated for the corresponding inode. If not, D-GRAID changes the mapping for the block to the correct home site. However, it is possible that a write to this block (that was made in the context of the new file) went to the old home site, and hence needs to be copied from its old physical location to the new location. Blocks that must be copied are added to a *pending copies list*; a background thread copies

the blocks to their new home site and frees the old physical locations when the copy completes.

6.1.5 Dealing with Imperfection

Another difficulty that arises in semantically-smart disks underneath typical file systems is that exact knowledge of the type of a dynamically-typed block is impossible to obtain, as discussed in Section 5. Thus, Alexander must handle incorrect type classification for data blocks (*i.e.*, file data, directory, and indirect blocks).

For example, D-GRAID must understand the contents of indirect blocks, because it uses the pointers therein to place a file's blocks onto its home site. However, due to lack of perfect knowledge, the fault-isolated placement of a file might be compromised (note that data loss or corruption is not an issue). Our goal in dealing with imperfection is thus to conservatively avoid it when possible, and eventually detect and handle it in all other cases.

Specifically, whenever a block construed to be an indirect block is written, we assume it is a valid indirect block. Thus, for every live pointer in the block, D-GRAID must take some action. There are two cases to consider. In the first case, a pointer could refer to an unmapped logical block. As mentioned before, D-GRAID then creates a new mapping in the home site corresponding to the inode to which the indirect block belongs. If this indirect block (and pointer) is valid, this mapping is the correct mapping. If this indirect block is misclassified (and consequently, the pointer invalid), D-GRAID detects that the block is free when it observes the data bitmap write, at which point the mapping is removed. If the block is allocated to a file before the bitmap is written, D-GRAID detects the reallocation during the inode write corresponding to the new file, creates a new mapping, and copies the data contents to the new home site (as discussed above).

In the second case, a potentially corrupt block pointer could point to an already mapped logical block. As discussed above, this type of block reuse results in a new mapping and copy of the block contents to the new home site. If this indirect block (and hence, the pointer) is valid, this new mapping is the correct one for the block. If instead the indirect block is a misclassification, Alexander wrongly copies over the data to the new home site. Note that the data is still accessible; however, the original file to which the block belongs, now has one of its blocks in the incorrect home site. Fortunately, this situation is transient, because once the inode of the file is written, D-GRAID detects this as a reallocation and creates a new mapping back to the original home site, thereby restoring its correct mapping. Files which are never accessed again are properly laid out by an infrequent sweep of inodes that looks for rare cases of improper layout.

Thus, without any optimizations, D-GRAID will eventually move data to the correct home site, thus preserving graceful degradation. However, to reduce the number of times such a misclassification occurs, Alexander makes an assumption about the contents of indirect blocks, specifically that they contain some number of valid unique pointers, or null pointers. Alexander can leverage this assumption to greatly reduce the number of misclassifications, by performing an integrity check on each supposed indirect block. The integrity check, which is reminiscent of work on conservative garbage collection [5], returns true if all the "pointers" (4-byte words in the block) point to valid data addresses within the volume and all non-null pointers are unique. Clearly, the set of blocks that pass this integrity check could still be corrupt if the data contents happened to exactly evade our conditions. However, a test run across the data blocks of our file system indicates that only a small fraction of data blocks (less than 0.1%) would pass the test; only those blocks that pass the test *and* are reallocated from a file data block to an indirect block would be misclassified.

6.1.6 Access-driven Diffusion

Another issue that D-GRAID must address is performance. Fault-isolated data placement improves availability but at the cost of performance. Data accesses to blocks of a large file, or, with directory-based grouping, to files within the same directory, are no longer parallelized. To improve performance, Alexander performs access-driven diffusion, monitoring block accesses to determine which are "hot", and then "diffusing" those blocks via replication across the disks of the system to enhance parallelism.

Access-driven diffusion can be achieved at both the logical and physical levels of a disk volume. In the logical approach, access to individual files is monitored, and those considered hot are diffused. However, per-file replication fails to capture sequentiality across multiple small files, for example, those in a single directory. Therefore we instead pursue a physical approach, in which Alexander replicates segments of the logical address space across the disks of the volume. Since file systems are good at allocating contiguous logical blocks for a single file, or to files in the same directory, replicating logical segments is likely to identify and exploit most common access patterns.

To implement access-driven diffusion, Alexander divides the logical address space into multiple segments, and during normal operation, gathers various statistics about the utilization and access patterns to each segment. A background thread selects logical segments that are likely to benefit most from access-driven diffusion and diffuses a copy across the drives of the system. Subsequent reads and writes first go to these replicas, with background updates sent to the original blocks. The imap entry for the block indicates which copy is up to date.

The amount of disk space to allocate to performance-oriented replicas presents an important policy decision. The initial policy that Alexander implements is to reserve a certain minimum amount of space (specified by the sys-

tem administrator) for these replicas, and then opportunistically use the free space available in the array for additional replication. This approach is similar to that used by AutoRAID for mirrored data [47], except that AutoRAID cannot identify data that is considered "dead" by the file system once written; in contrast, D-GRAID can use semantic knowledge to identify which blocks are free.

6.2 Live-block Recovery

To implement live-block recovery, D-GRAID must understand which blocks are live. This knowledge must be correct in that no block that is live is considered dead, as that would lead to data loss. Alexander tracks this information by observing bitmap and data block traffic. Bitmap blocks tell us the liveness state of the file system that has been reflected to disk. However, due to reordering and delayed updates, it is not uncommon to observe a write to a data block whose corresponding bit has not yet been set in the data bitmap. To account for this, D-GRAID maintains a duplicate copy of all bitmap blocks, and whenever it sees a write to a block, sets the corresponding bit in the local copy of the bitmap. The duplicate copy is synchronized with the file system copy when the data bitmap block is written by the file system. This *conservative bitmap table* thus reflects a superset of all live blocks in the file system, and can be used to perform live-block recovery. Note that we assume the pre-allocation state of the bitmap will not be written to disk after a subsequent allocation; the locking in Linux and other modern systems already ensures this. Though this technique guarantees that a live block is never classified as dead, it is possible for the disk to consider a block live far longer than it actually is. This situation would arise, for example, if the file system writes deleted blocks to disk.

To implement live-block recovery, Alexander simply uses the conservative bitmap table to build a list of blocks which need to be restored. Alexander then proceeds through the list and copies all live data onto the hot spare.

6.3 Other Aspects of Alexander

There are a host of other aspects of the implementation that are required for a successful prototype but that we cannot discuss at length due to space limitations. For example, we found that preserving the logical contiguity of the file system was important in block allocation, and thus developed mechanisms to enable such placement. Directory-based grouping also requires more sophistication in the implementation, to handle the further deferral of writes until a parent directory block is written. "Just in time" block allocation prevents misclassified indirect blocks from causing spurious physical block allocation. Deferred list management introduces some tricky issues when there is not enough memory. Alexander also preserves "sync" semantics by not returning success on inode block writes until deferred block writes that were waiting on the inode complete. There are a number of structures that Alexander maintains, such as the imap, that must be reliably committed to disk and preferably, for good performance, buffered in a small amount of non-volatile RAM.

The most important component that is missing from Alexander is the decision on which "popular" (read-only) directories such as `/usr/bin` to replicate widely, and when to do so. Although Alexander contains the proper mechanisms to perform such replication, the policy space remains unexplored. However, our initial experience indicates that a simple approach based on monitoring frequency of inode access time updates may likely be effective. An alternative approach allows administrators to specify directories that should be treated in this manner.

One interesting issue that required a change from our design was the behavior of Linux ext2 under partial disk failure. When a process tries to read a data block that is unavailable, ext2 issues the read and returns an I/O failure to the process. When the block becomes available again (*e.g.*, after recovery) and a process issues a read to it, ext2 will again issue the read, and everything works as expected. However, if a process tries to open a file whose inode is unavailable, ext2 marks the inode as "suspicious" and will never again issue an I/O request to the inode block, even if Alexander has recovered the block. To avoid a change to the file system and retain the ability to recover failed inodes, Alexander replicates inode blocks as it does namespace meta-data, instead of collocating them with the data blocks of a file.

6.4 Alexander the FAT

Overall, we were surprised by the many similarities we found in implementing D-GRAID underneath ext2 and VFAT. For example, VFAT also overloads data blocks, using them as either user data blocks or directories; hence Alexander must defer classification of those blocks in a manner similar to the ext2 implementation.

However, there were a few instances where the VFAT implementation of D-GRAID differed in interesting ways from the ext2 version. For example, the fact that all pointers of a file are located in the file allocation table made a number of aspects of D-GRAID much simpler to implement; in VFAT, there are no indirect pointers to worry about. We also ran across the occasional odd behavior in the Linux implementation of VFAT. For example, Linux would write to disk data blocks that were allocated but then freed, avoiding an obvious and common file system optimization. Although this was more indicative of the untuned nature of the Linux implementation, it served as yet another indicator of how semantic disks must be wary of any assumptions they make about file system behavior.

7 Evaluating Alexander

We now present a performance evaluation of Alexander. We focus primarily on the Linux ext2 variant, but also

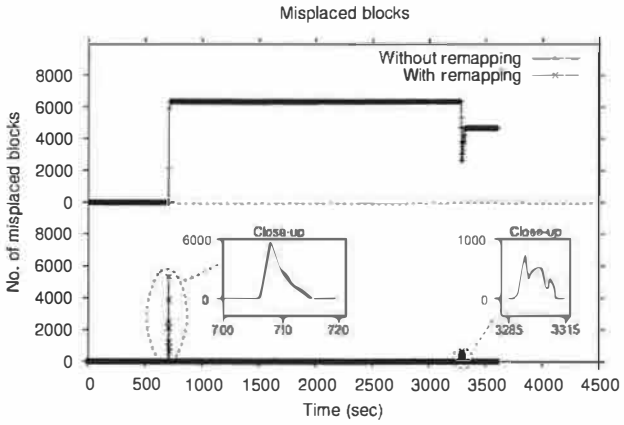


Figure 5: **Errors in Placement.** *The figure plots the number of blocks wrongly laid out by Alexander over time, while running a busy hour of the HP Trace. The experiment was run over 4 disks, and the total number of blocks accessed in the trace was 418000.*

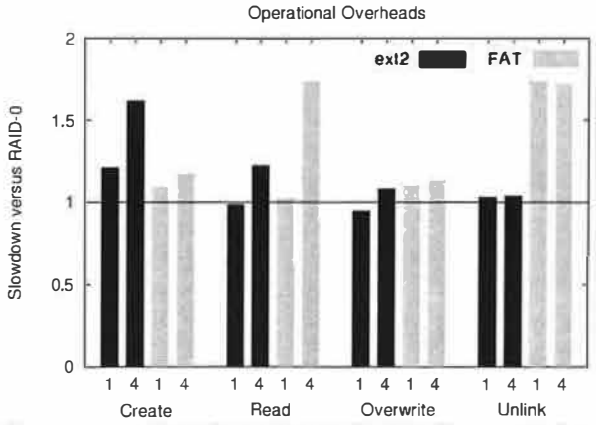


Figure 6: **Time Overheads.** *The figure plots the time overheads observed on D-GRAID Level 0 versus RAID Level 0 across a series of microbenchmarks. The tests are run on 1 and 4 disk systems. In each experiment, 3000 operations were enacted (e.g., 3000 file creations), with each operation on a 64 KB file.*

include some baseline measurements of the VFAT system. We wish to answer the following questions:

- *Does Alexander work correctly?*
- *What time overheads are introduced?*
- *How effective is access-driven diffusion?*
- *How fast is live-block recovery?*
- *What overall benefits can we expect from D-GRAID?*
- *How complex is the implementation?*

7.1 Platform

The Alexander prototype is constructed as a software RAID driver in the Linux 2.2 kernel. File systems mount the pseudo-device and use it as if it were a normal disk. Our environment is excellent for understanding many of the issues that would be involved in the construction of a "real" hardware D-GRAID system; however, it is also limited in the following ways. First, and most importantly, Alexander runs on the same system as the host OS and applications, and thus there is interference due to competition for resources. Second, the performance characteristics of the microprocessor and memory system may be different than what is found within an actual RAID system. In the following experiments, we utilize a 550 MHz Pentium III and four 10K-RPM IBM disks.

Does Alexander work correctly? Alexander is more complex than simple RAID systems. To ensure that Alexander operates correctly, we have put the system through numerous stress tests, moving large amounts of data in and out of the system without problems. We have also extensively tested the corner cases of the system, pushing it into situations that are difficult to handle and making sure that the system degrades gracefully and recovers as expected. For example, we repeatedly crafted microbenchmarks to stress the mechanisms for detecting block reuse and for handling imperfect information about dynamically-typed blocks. We have also constructed benchmarks that write user data blocks to disk that contain "worst case" data, *i.e.*, data that appears to be valid directory entries or indirect pointers. In all cases, Alexander was able to detect which blocks were indirect blocks and move files and directories into their proper fault-isolated locations.

To verify that Alexander places blocks on the appropriate disk, we instrumented the file system to log block allocations. In addition, Alexander logs events of interest such as assignment of a home site for an inode, creation of a new mapping for a logical block, re-mapping of blocks to a different homesite and receipt of logical writes from the file system. To evaluate the behavior of Alexander on a certain workload, we run the workload on Alexander, and obtain the time-ordered log of events that occurred at the file system and Alexander. We then process this log off-line and look for the number of blocks wrongly laid out at any given time.

We ran this test on a few hours of the HP Traces, and found that in many of the hours we examined, the number of blocks that were misplaced even temporarily was quite low, often less than 10 blocks. We report detailed results for one such hour of the trace where we observed the greatest number of misplaced blocks, among the hours we examined. Figure 5 shows the results.

The figure has two parts. The bottom part shows the normal operation of Alexander, with the capability to react to block reuse by remapping (and copying over) blocks to the correct homesite. As the figure shows, Alexander is able to quickly detect wrongly placed blocks and remap them appropriately. Further, the number of such blocks misplaced temporarily is only about 1% of the total number of blocks accessed in the trace. The top part of the figure shows the number of misplaced blocks for the same experiment, but assuming that the remapping did not occur. As can be expected, those delinquent blocks remain misplaced. The dip towards the end of the trace occurs

	Run-time (seconds)	Blocks Written		
		Total	Meta data	Unique
RAID-0	69.25	101297	–	–
D-GRAID$_1$	61.57	93981	5962	1599
D-GRAID$_2$	66.50	99416	9954	3198
D-GRAID$_3$	73.50	101559	16976	4797
D-GRAID$_4$	78.79	113222	23646	6396

Table 2: **Performance on postmark.** *The table compares the performance of D-GRAID Level 0 with RAID-0 on the Postmark benchmark. Each row marked D-GRAID indicates a specific level of metadata replication. The first column reports the benchmark run-time and the second column shows the number of disk writes incurred. The third column shows the number of disk writes that were to metadata blocks, and the fourth column indicates the number of unique metadata blocks that are written. The experiment was run over 4 disks.*

because some of the misplaced blocks are later assigned to a file in that homesite itself, accidentally correcting the original misplacement.

What time overheads are introduced? We now explore the time overheads that arise due to semantic inference. This primarily occurs when new blocks are written to the file system, such as during file creation. Figure 6 shows the performance of Alexander under a simple microbenchmark. As can be seen, allocating writes are slower due to the extra CPU cost involved in tracking fault-isolated placement. Reads and overwrites perform comparably with RAID-0. The high unlink times of D-GRAID on FAT is because FAT writes out data pertaining to deleted files, which have to be processed by D-GRAID as if it were newly allocated data. Given that the implementation is untuned and the infrastructure suffers from CPU and memory contention with the host, we believe that these are worst case estimates of the overheads.

Another cost of D-GRAID that we explore is the overhead of metadata replication. For this purpose, we choose Postmark [25], a metadata intensive file system benchmark. We slightly modified Postmark to perform a `sync` before the deletion phase, so that all metadata writes are accounted for, making it a pessimistic evaluation of the costs. Table 2 shows the performance of Alexander under various degrees of metadata replication. As can be seen from the table, synchronous replication of metadata blocks has a significant effect on performance for metadata intensive workloads (the file sizes in Postmark range from 512 bytes to 10 KB). Note that Alexander performs better than default RAID-0 for lower degrees of replication because of better physical block allocation; since ext2 looks for a contiguous free chunk of 8 blocks to allocate a new file, its layout is sub-optimal for small files.

The table also shows the number of disk writes incurred during the course of the benchmark. The percentage of extra disk writes roughly accounts for the difference in performance between different replication levels, and these extra writes are mostly to metadata blocks. However, when we count the number of unique physical writes to metadata blocks, the absolute difference between different replication levels is small. This suggests that lazy propagation of updates to metadata block replicas, perhaps during idle time or using freeblock scheduling, can greatly reduce the performance difference, at the cost of added complexity. For example, with lazy update propagation (*i.e.*, if the replicas were updated only once), D-GRAID$_4$ would incur only about 4% extra disk writes.

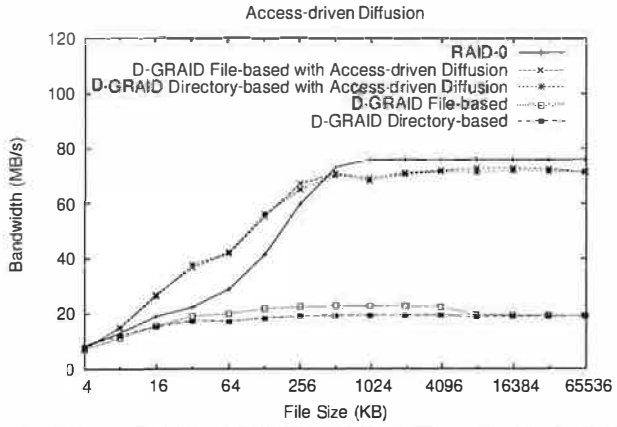


Figure 7: **Access-driven Diffusion.** *The figure presents the performance of D-GRAID Level 0 and standard RAID-0 under a sequential workload. In each experiment, a number of files of size x are read sequentially, with the total volume of data fixed at 64 MB. D-GRAID performs better for smaller files due to better physical block layout.*

We also played back a portion of the HP traces for 20 minutes against a standard RAID-0 system and D-GRAID over four disks. The playback engine issues requests at the times specified in the trace, with an optional speedup factor; a speedup of 2× implies the idle time between requests was reduced by a factor of two. With speedup factors of 1× and 2×, D-GRAID delivered the same per-second operation throughput as RAID-0, utilizing idle time in the trace to hide its extra CPU overhead. However, with a scaling factor of 3×, the operation throughput lagged slightly behind, with D-GRAID showing a slowdown of up to 19.2% during the first one-third of the trace execution, after which it caught up due to idle time.

How effective is access-driven diffusion? We now show the benefits of access-driven diffusion. In each trial of this experiment, we perform a set of sequential file reads, over files of increasing size. We compare standard RAID-0 striping to D-GRAID with and without access-driven diffusion. Figure 7 shows the results of the experiment.

As we can see from the figure, without access-driven diffusion, sequential access to larger files run at the rate of a single disk in the system, and thus do not benefit from the potential parallelism. With access-driven diffusion, performance is much improved, as reads are directed

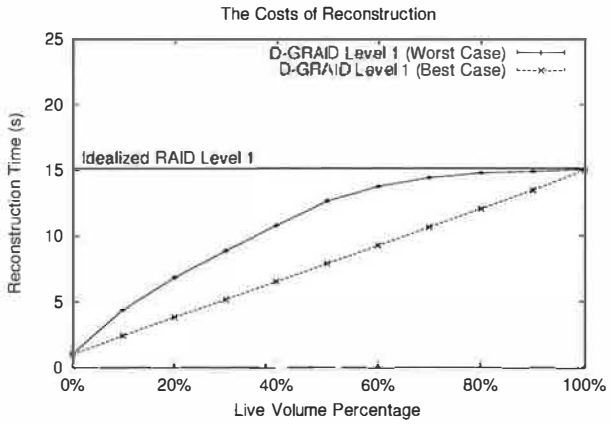


Figure 8: **Live-block Recovery.** *The figure shows the time to recover a failed disk onto a hot spare in a D-GRAID Level 1 (mirrored) system using live-block recovery. Two lines for D-GRAID are plotted: in the worst case, live data is spread across the entire 300 MB volume, whereas in the best case it is compacted into the smallest contiguous space possible. Also plotted is the recovery time of an idealized RAID Level 1.*

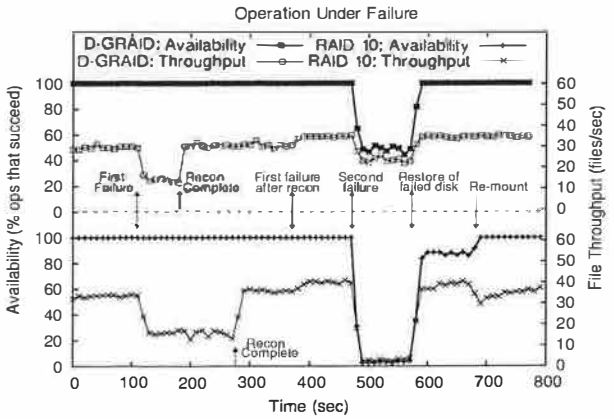


Figure 9: **Availability Profile.** *The figure shows the operation of D-GRAID Level 1 and RAID 10 under failures. The 3 GB array consists of 4 data disks and 1 hot spare. After the first failure, data is reconstructed onto the hot spare, D-GRAID recovering much faster than RAID 10. When two more failures occur, RAID 10 loses almost all files, while D-GRAID continues to serve 50% of its files. The workload consists of read-modify-writes of 32 KB files randomly picked from a 1.3 GB working set.*

to the diffused copies across all of the disks in the system. Note that in the latter case, we arrange for the files to be already diffused before the start of the experiment, by reading them a certain threshold number of times. Investigating more sophisticated policies for when to initiate access-driven diffusion is left for future work.

How fast is live-block recovery? We now explore the potential improvement seen with live-block recovery. Figure 8 presents the recovery time of D-GRAID while varying the amount of live file system data.

The figure plots two lines: worst case and best case live-block recovery. In the worst case, live data is spread throughout the disk, whereas in the best case it is compacted into a single portion of the volume. From the graph, we can see that live-block recovery is successful in reducing recovery time, particularly when a disk is less than half full. Note also the difference between worst case and best case times; the difference suggests that periodic disk reorganization [41] could be used to speed recovery, by moving all live data to a localized portion.

What overall benefits can we expect from D-GRAID? We next demonstrate the improved availability of Alexander under failures. Figure 9 shows the availability and performance observed by a process randomly accessing whole 32 KB files, running above D-GRAID and RAID-10. To ensure a fair comparison, both D-GRAID and RAID-10 limit their reconstruction rate to 10 MB/s.

As the figure shows, reconstruction of the 3 GB volume with 1.3 GB live data completes much faster (68 s) in D-GRAID compared to RAID-10 (160 s). Also, when the extra second failure occurs, the availability of RAID-10 drops to near zero, while D-GRAID continues with about 50 % availability. Surprisingly, after restore, RAID-10 still fails on certain files; this is because Linux does not retry inode blocks once they fail. A remount is required before RAID-10 returns to full availability.

How complex is the implementation? We briefly quantify the implementation complexity of Alexander. Table 3 shows the number of C statements required to implement the different components of Alexander. From the table, we can see that the core file system inferencing module for ext2 requires only about 1200 lines of code (counted with number of semicolons), and the core mechanisms of D-GRAID contribute to about 2000 lines of code. The rest is spent on a hash table, AVL tree and wrappers for memory management. Compared to the tens of 1000's of lines of code already comprising modern array firmware, we believe that the added complexity of D-GRAID is not that significant.

8 Discussion

In this section, we first compare our semantic-disk based approach to alternative methods of implementing D-GRAID, and then discuss some possible concerns about the commercial feasibility of such semantic disk systems.

8.1 Alternative Approaches

Our semantic disk based approach is one of few different ways of implementing D-GRAID, each with its own trade-offs. Similar to modern processors that innovate beneath unchanged instruction sets, a semantic disk level implementation facilitates ease of deployment and inter-operability with unchanged client infrastructure, perhaps making it more pragmatic. The cost of this approach, however, is the complexity in rediscovering semantic knowledge and being tolerant to inaccuracies.

An alternative approach is to change the interface between file systems and storage, to convey richer information across both layers. For instance, the storage system could expose failure boundaries to the file system [9], and

	Semicolons	Total
D-GRAID Generic		
Setup + fault-isolated placement	1726	3557
Physical block allocation	322	678
Access driven diffusion	108	238
Mirroring + live block recovery	248	511
Internal memory management	182	406
Hashtable/Avl tree	724	1706
File System Specific		
SDS Inferencing: ext2	1252	2836
SDS Inferencing: VFAT	630	1132
Total	5192	11604

Table 3: **Code size for Alexander implementation.** *The number of lines of code needed to implement Alexander is shown. The first column shows the number of semicolons and the second column shows the total number of lines, including white-spaces and comments.*

then the file system could explicitly allocate blocks in a fault-isolated manner. Alternatively, the file system could tag each write with a logical fault-container ID, which can then be used by the storage system to implement fault-isolated data placement. These techniques, while being intrusive on existing infrastructure and software base, are conceivably less complex than our approach.

Object-based storage [16] is one such new interface being considered, which makes the file boundaries more visible at the storage layer. However, even with an object-based interface, semantically-smart technology might still be relevant to discover relationships across objects; for instance inferring that a directory object points to a set of file objects which need to collocated.

8.2 Commercial Feasibility

By definition, D-GRAID and other semantically-smart storage systems have more detailed knowledge of the file system that is using them. Embedding a higher degree of functionality within the storage system leads to some concerns on the commercial feasibility of such systems.

The first concern that arises is that placing semantic knowledge within the disk system ties the disk system too intimately to the file system above. For example, if the file system's on-disk structure changes, the storage system may have to change as well. We believe this issue is not likely to be problematic. On-disk formats evolve slowly, for reasons of backwards compatibility. For example, the basic structure of FFS-based file systems has not changed since its introduction in 1984, a period of almost twenty years [28]; the Linux ext2 file system, introduced in roughly 1994, has had the exact same layout for its lifetime. finally, the ext3 journaling file system [45] is backwards compatible with ext2 on-disk layout and the new extensions to the FreeBSD file system [10] are backwards compatible as well. We also have evidence that storage vendors are already willing to maintain and support software specific to a file system; for example, the EMC Symmetrix storage system [11] comes with software that can understand the format of most common file systems.

The second concern is that the storage system needs semantic knowledge for each file system with which it interacts. Fortunately, there are not a large number of file systems that would need to be supported to cover a large fraction of the usage population. If such a semantic storage system is used with a file system that it does not support, the storage system could detect that the file system does not conform to its expectations and tum off its special functionality (*e.g.*, in the case of D-GRAID, revert to normal RAID layout). Such detection can be done by simple techniques such as observing the file system identifier in the partition table.

One final concern that arises is that too much processing will be required within the disk system. We do not believe this to be a major issue, because of the general trend of increasing disk system intelligence [1, 37]; as processing power increases, disk systems are likely to contain substantial computational abilities. Indeed, modem storage arrays already exhibit the fruits of Moore's Law; for example, the EMC Symmetrix storage server can be configured with up to 80 processors and 64 GB of RAM [11].

9 Related Work

D-GRAID draws on related work from a number of different areas, including distributed file systems and traditional RAID systems. We discuss each in turn.

Distributed File Systems: Designers of distributed file systems have long ago realized the problems that arise when spreading a directory tree across different machines in a system. For example, Walker *et al.* discuss the importance of directory namespace replication within the Locus distributed system [35]. The Coda mobile file system also takes explicit care with regard to the directory tree [27]. Specifically, if a file is cached, Coda makes sure to cache every directory up to the root of the directory tree. By doing so, Coda can guarantee that a file remains accessible should a disconnection occur. Perhaps an interesting extension to our work would be to reconsider host-based in-memory caching with availability in mind. Also, Slice [3] tries to route namespace operations for all files in a directory to the same server.

More recently, work in wide-area file systems has also re-emphasized the importance of the directory tree. For example, the Pangaea file system aggressively replicates the entire tree up to the root on a node when a file is accessed [42]. The Island-based file system also points out the need for "fault isolation" but in the context of wide-area storage systems; their "one island principle" is quite similar to fault-isolated placement in D-GRAID [24].

Finally, p2p systems such as PAST that place an entire file on a single machine have similar load balancing issues [40]. However, the problem is more difficult in the p2p space due to the constraints of file placement; block

migration is much simpler in a centralized storage array.

Traditional RAID Systems: We also draw on the long history of research in classic RAID systems. From AutoRAID [47] we learned both that complex functionality could be embedded within a modern storage array, and that background activity could be utilized successfully in such an environment. From AFRAID [43], we learned that there could be a flexible trade-off between performance and reliability, and the value of delaying updates.

Much of RAID research has focused on different redundancy schemes. While early work stressed the ability to tolerate single-disk failures [4, 32, 33], later research introduced the notion of tolerating multiple-disk failures within an array [2, 6]. We stress that our work is complementary to this line of research; traditional techniques can be used to ensure full file system availability up to a certain number of failures, and D-GRAID techniques ensure graceful degradation under additional failures. A related approach is parity striping [18] which stripes only the parity and not data; while this would achieve some fault isolation, the layout is still oblivious of the semantics of the data; blocks will have the same level of redundancy irrespective of their importance (*i.e.*, meta-data vs data), so multiple failures could still make the entire file system inaccessible. A number of earlier works also emphasize the importance of hot sparing to speed recovery time in RAID arrays [21, 29, 32]. Our work on semantic recovery is also complementary to those approaches.

Finally, note that term "graceful degradation" is sometimes used to refer to the performance characteristics of redundant disk systems under failure [22, 36]. This type of graceful degradation is different from what we discuss in this paper; indeed, none of those systems continues operation when an unexpected number of failures occurs.

10 Conclusions

> "A robust system is one that continues to operate (nearly) correctly in the presence of some class of errors" *Robert Hagmann [20]*

D-GRAID turns the simple binary failure model found in most storage systems into a continuum, increasing the availability of storage by continuing operation under partial failure and quickly restoring live data after a failure does occur. In this paper, we have shown the potential benefits of D-GRAID, established the limits of semantic knowledge, and have shown how a successful D-GRAID implementation can be achieved despite these limits. Through simulation and the evaluation of a prototype implementation, we have found that D-GRAID can be built underneath a standard block-based interface, without any file system modification, and that it delivers graceful degradation and live-block recovery, and, through access-driven diffusion, good performance.

We conclude with a discussions of the lessons we have learned in the process of implementing D-GRAID:

- **Limited knowledge within the disk does not imply limited functionality.** One of the main contributions of this paper is a demonstration of both the limits of semantic knowledge, as well as the "proof" via implementation that despite such limitations, interesting functionality can be built inside of a semantically-smart disk system. We believe any semantic disk system must be careful in its assumptions about file system behavior, and hope that our work can guide others who pursue a similar course.
- **Semantically-smart disks would be easier to build with some help from above.** Because of the way file systems reorder, delay, and hide operations from disks, reverse engineering exactly what they are doing at the SCSI level is difficult. We believe that small modifications to file systems could substantially lessen this difficulty. For example, if the file system could inform the disk whenever it believes the file system structures are in a consistent on-disk state, many of the challenges in the disk would be lessened. This is one example of many small alterations that could ease the burden of semantic disk development.
- **Semantically-smart disks stress file systems in unexpected ways.** File systems were not built to operate on top of disks that behave as D-GRAID does; specifically, they may not behave particularly well when part of a volume address space becomes unavailable. Perhaps because of its heritage as an OS for inexpensive hardware, Linux file systems handle unexpected conditions fairly well. However, the exact model for dealing with failure is inconsistent: data blocks could be missing and then reappear, but the same is not true for inodes. As semantically-smart disks push new functionality into storage, file systems would likely have to evolve to accommodate them.
- **Detailed traces of workload behavior are invaluable.** Because of the excellent level of detail available in the HP traces [38], we were able to simulate and analyze the potential of D-GRAID under realistic settings. Many other traces do not contain per-process information, or anonymize file references to the extent that pathnames are not included in the trace, and thus we could not utilize them in our study. One remaining challenge for tracing is to include user data blocks, as semantically-smart disks may be sensitive to the contents. However, the privacy concerns that such a campaign would encounter may be too difficult to overcome.

Acknowledgments

We would like to thank Anurag Acharya, Erik Riedel, Yasushi Saito, John Bent, Nathan Burnett, Timothy Denehy, Brian Forney, Florentina Popovici, and Lakshmi Bairavasundaram for their insightful comments on earlier drafts of the paper. We also would like to thank Richard Golding for his excellent shepherding, and the anonymous reviewers for their thoughtful suggestions, many of which have greatly improved the content of this paper. Finally, we

thank the Computer Systems Lab for providing a terrific environment for computer science research.

This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM, EMC, and the Wisconsin Alumni Research Foundation.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: programming model, algorithms and evaluation. In *ASPLOS VIII*, San Jose, CA, October 1998.

[2] G. A. Alvarez, W. A. Burkhard, and F. Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *ISCA '97*, pages 62–72, 1997.

[3] D. Anderson, J. Chase, and A. Vahdat. Interposed Request Routing for Scalable Network Storage. *ACM Transactions on Computer Systems*, 20(1), February 2002.

[4] D. Bitton and J. Gray. Disk shadowing. In *VLDB 14*, pages 331–338, Los Angeles, CA, August 1988.

[5] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[6] W. Burkhard and J. Menon. Disk Array Storage System Reliability. In *FTCS-23*, pages 432–441, Toulouse, France, June 1993.

[7] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *SOSP '95*, December 1995.

[8] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[9] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *USENIX '02*, June 2002.

[10] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD. In *FREENIX '02*, Monterey, CA, June 2002.

[11] EMC Corporation. Symmetrix Enterprise Information Storage Systems. http://www.emc.com, 2002.

[12] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *USENIX Winter '92*, January 1992.

[13] G. R. Ganger. Blurring the Line Between Oses and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[14] G. R. Ganger, M. K. McKusick, C. A. Soules, and Y. N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM TOCS*, 18(2), May 2000.

[15] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt. Disk Subsystem Load Balancing: Disk Striping vs. Conventional Data Placement. In *HICSS '93*, 1993.

[16] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *ASPLOS VIII*, October 1998.

[17] J. Gray. Why Do Computers Stop and What Can We Do About It? In *6th International Conference on Reliability and Distributed Databases*, June 1987.

[18] J. Gray, B. Horst, and M. Walker. Parity Striping of Disc Arrays: Low-cost Reliable Storage with Acceptable Throughput. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB 16)*, pages 148–159, Brisbane, Australia, August 1990.

[19] S. D. Gribble. Robustness in Complex Systems. In *HotOS VIII*, Schloss Elmau, Germany, May 2001.

[20] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, November 1987.

[21] M. Holland, G. Gibson, and D. Siewiorek. Fast, on-line failure recovery in redundant disk arrays. In *FTCS-23*, France, 1993.

[22] H.-I. Hsiao and D. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *6th International Data Engineering Conference*, 1990.

[23] IBM. ServeRAID - Recovering from multiple disk failures. http://www.pc.ibm.com/qtechinfo/MIGR-39144.html, 2001.

[24] M. Ji, E. Felten, R. Wang, and J. P. Singh. Archipelago: An Island-Based File System For Highly Available And Scalable Internet Services. In *4th USENIX Windows Symposium*, August 2000.

[25] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., Oct 1997.

[26] K. Keeton and J. Wilkes. Automating data dependability. In *Proceedings of the 10th ACM-SIGOPS European Workshop*, pages 93–100, Saint-Emilion, France, September 2002.

[27] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM TOCS*, 10(1), February 1992.

[28] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM TOCS*, 2(3):181–197, August 1984.

[29] J. Menon and D. Mattson. Comparison of Sparing Alternatives for Disk Arrays. In *ISCA '92*, Gold Coast, Australia, May 1992.

[30] Microsoft Corporation. http://www.microsoft.com/hwdev/, December 2000.

[31] C. U. Orji and J. A. Solworth. Doubly Distorted Mirrors. In *SIGMOD '93*, Washington, DC, May 1993.

[32] A. Park and K. Balasubramanian. Providing fault tolerance in parallel secondary storage systems. Technical Report CS-TR-057-86, Princeton, November 1986.

[33] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, June 1988.

[34] D. A. Patterson. Availability and Maintainability >> Performance: New Focus for a New Century. Key Note at FAST '02, January 2002.

[35] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. In *SOSP '81*, December 1981.

[36] A. L. N. Reddy and P. Banerjee. Gracefully Degradable Disk Arrays. In *FTCS-21*, pages 401–408, Montreal, Canada, June 1991.

[37] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large Scale Data Mining and Multimedia. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB 24)*, New York, New York, August 1998.

[38] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *FAST '02*, pages 14–29, Monterey, CA, January 2002.

[39] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[40] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *SOSP '01*, Banff, Canada, October 2001.

[41] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-91-156, Hewlett Packard Laboratories, 1991.

[42] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *OSDI '02*, Boston, MA, December 2002.

[43] S. Savage and J. Wilkes. AFRAID — A Frequently Redundant Array of Independent Disks. In *USENIX 1996*, pages 27–39, San Diego, CA, January 1996.

[44] M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *FAST '03*, San Francisco, CA, March 2003.

[45] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX '02*, Monterey, CA, June 2002.

[46] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *OSDI '99*, New Orleans, LA, February 1999.

[47] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[48] J. L. Wolf. The Placement Optimization Problem: A Practical Solution to the Disk File Assignment Problem. In *SIGMETRICS '89*, pages 1–10, Berkeley, CA, May 1989.

Polus : Growing Storage QoS Management Beyond a "Four-year Old Kid"

Sandeep Uttamchandani Kaladhar Voruganti Sudarshan Srinivasan*
John Palmer David Pease

{sandeepu,kaladhar,smsriniv,jdp,dpease}@us.ibm.com

IBM Almaden Research Center,
650 Harry Road, San José, California 95120, USA

Abstract

Policy-based storage management has been advertised as the silver bullet to overcome the complexity that limits the amount of storage that can be managed by system administrators. Key to this approach are: a mechanism to specify quality of service (QoS) goals; a canonical virtual model of storage devices and operations; and the mapping of the high level QoS goals to low level storage device actions. In spite of prior research and industrial standards the latter problem results in complex, manual, error-prone processes that burden system administrators and prevent the widespread acceptance of policy-based storage management. This paper proposes the Polus framework which specifically addresses this open problem.

Polus removes the need for system administrators to write code that maps the QoS goals to low level system actions. Instead, it generates this mapping code by using a combination of rule-of-thumb specification mechanism, a reasoning engine and a learning engine to change the implementation paradigm of policy-based storage management. This paper also provides a quantitative analysis of the Polus framework within the context of a storage area network (SAN) file system to verify the feasibility of this new approach.

1 Introduction

Capacity planning, application/storage performance management, backup/restore operations, configuration management, security, and availability analysis are some of the key storage management responsibilities of a system administrator. Typically, storage administrators write scripts that automate many of these storage management tasks. As the number of business service level agreements, department policies, QoS goals, storage devices, protocols, applications, and users increases, it becomes difficult for system administrators to ensure performance, provisioning, availability and security goals by using ad hoc script writing approaches . Thus, systems management has been identified as one of the most important research areas by many leading researchers [9, 26]. Storage vendors are trying to add sophisticated systems management functionality into databases, file systems, storage controllers, storage resource managers, storage area network managers, capacity planning managers and other storage management software. The major focus of these products is to reduce management complexity by allowing a system administrator to specify high level QoS goals with respect to expected performance,availability, provisioning, and security, and to automatically transform these high level QoS goals into low level system actions.

Currently, this transformation process is built using the policy-based paradigm, where policies are specified as collection of rules that are in the ECA format (Event-Condition-Action) [14]. Rules define how the system behaves for different possible system states and goal values. At run-time, the management module simply invokes the rule that is applicable based on the event and system condition. Even though goal based storage management approach has been advocated as the silver bullet that can help to reduce the management complexity for system administrators, this approach has not gained much traction because current policy management frameworks are providing support for only simple and trivial storage management scenarios.

1.1 Problem Statement

Design of high level QoS goals to low level storage actions transformation mechanisms in management software is done by experts with many years of prior experience as system architects and administrators. However, even the experts are encountering the following types of problems while designing robust storage management systems:

Complexity: The level of details, required to write the specifications is non-trivial. ECA rules are written as a certain storage management action being taken when a system observable violates a predetermined threshold. The transformation code is specified as ECA rules (e.g. if *throughput_goal_violated* AND *Sequential/Random_ratio* > 1, then increase data

*IBM Summer intern, 2003. Affiliation: Department of Computer Science, University of Illinois at Urbana Champaign. E-mail:smsriniv@cs.uiuc.edu

prefetching size by 20%) where actions are taken upon the violation of threshold values. It is difficult for the composers of ECA rules to: (1) choose which combination of system parameters to observe from a large set of possible observables; (2) determine appropriate threshold values after considering the interaction of a large set of system variables; and (3) select a specific corrective action from the large set of competing options. As the number of users, storage devices, storage management actions, and service level agreements increase, it becomes computationally exhaustive for system administrators or storage management tool developers to consider all the alternatives.

Brittleness: It is difficult for vendors to provide prepackaged transformation code with their products because this code becomes brittle with respect to changing system configurations, user workloads, and department/business constraints. It is difficult for the storage management vendors to envision all of the potential use case scenarios ahead of time, and thus, many of the current storage management solutions provide work-flow environments which, in turn, pass the responsibility of transforming high level QoS goals (via work-flow scripts) to an organization's system administrators and infrastructure planners.

To summarize, we restate the discussion-panel conclusion in FAST'03: Existing storage management frameworks are like a "four-year old kid"- "They mess up more than they are actually useful."

1.2 Bird's eye view and Contributions of Polus Framework

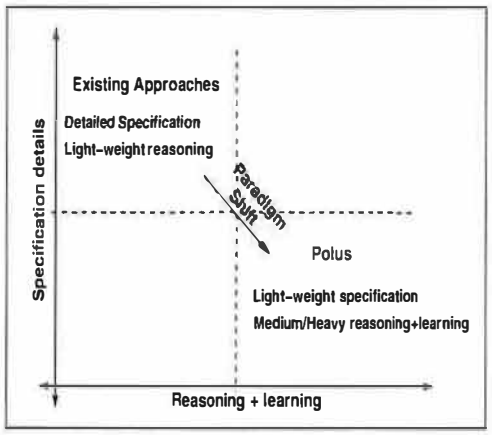


Figure 1. An innovative approach for QoS management

This paper proposes the Polus framework that tries to take away the complexity of writing policy mapping code from human experts and moves it to a combination of reasoning and learning engines (as shown in Figure 1). The Polus framework addresses the complexity and brittleness problems described in section 1.1, in the following manner.

In Polus, as shown in Figure 2, the system administrator inputs knowledge in the form of rules of thumb. For example, "To invoke Prefetch action *requires* memory", "Invoke Prefetch action *requires* the workload to be Sequential", "Prefetch action *improves* throughput." The generic form of the specification is (<*Relationship* >[*Type of relationship*]) where the *Relationship* is between *actions* and *resources, workload characteristics, system behavior*. The optional *Type of relationship* gives a hint about the nature of the relationship (e.g., *improves, requires*). It should be noted that the rule-of-thumb specifications do not require the system administrator to quantify the threshold values for actions, observables, workload characteristics and resources. In addition, he does not have to spell-out the details of the action to be invoked for a given state of resources, workloads, and observables.

The relationships defined in the rule-of-thumb specifications are quantified by the use of a learning engine. The system management actions, the state of the system resources/workload characteristics when the a particular action was taken, and the current values of the observables (e.g., throughput, latency, etc.) are monitored and stored as part of the knowledge base. The learning engine uses this monitored information to predict and quantify the relationships described in the specifications; for example, "Prefetching improves throughput when available memory is greater than 20 percent", and "Use prefetching when Sequential/Random ratio is greater than 0.4." It is important to note that the rule-of-thumb specifications help to prune the number of variables used in the interpolation function, which in turn helps improve the convergence rate of the learning function. For example, while interpolating relationships of the prefetch action, the system is not required to take into account observables related to security. In the current implementation of Polus, the learning engine does not discover additional relationships (apart from those in the specifications) and also assumes that the hints in the specifications are correct. In the future, these assumptions will be addressed using learning approaches such as "bagging" [5].

When a particular QoS goal is violated in the system, the Polus reasoning engine is invoked. The semantics of the reasoning engine are expressed in first-order predicate calculus and are similar to the thought-process that is implicit in ECA rules. For example, an action (x) can be invoked only if the resource required (*precondition*)

for its invocation are available in the current-state (cs). The current-state is defined in terms of values of resources, workload characteristics, and observables. This is expressed in first-order predicate calculus as:

$$\forall\, x, invoke(x, cs) \Rightarrow$$
$$(available(cs) > precondition(x))$$

Using the current-state as input and the information (i.e., combination of specifications and learning) in the knowledge base as facts, the reasoning engine derives the actions to be invoked at run-time. For example, when the reasoning engine tries to invoke the prefetch action the *invoke* function is instantiated as $invoke(prefetch, cs)$ which will be true if

$$(available(cs) > precondition(prefetch))$$

In this predicate, cs is unified with the values that were passed as input to the reasoning engine, and the information of the prefetch action is retrieved from the knowledge base.

It is important to note that combinations of declarative specification and predicate calculus (as done in Polus) are the basis of the well-known field of logic based programming [16, 10]. Polus is applying and extending these concepts for the domain of storage management.

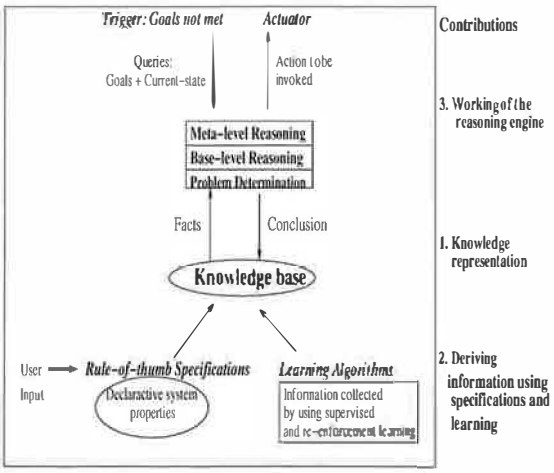


Figure 2. Contributions of Polus

1.3 Paper Organization

Sections 2 and 3 provide the details of ECA and Polus respectively. Section 4 describes the experimental framework. Section 5 presents the experiment results. Section 6 contains a discussion of the experiment results. A survey of previous expert systems, policy frameworks and storage management solutions is provided in section 7. Finally, section 8 presents our concluding remarks.

2 Background: QoS Management Using the ECA Approach

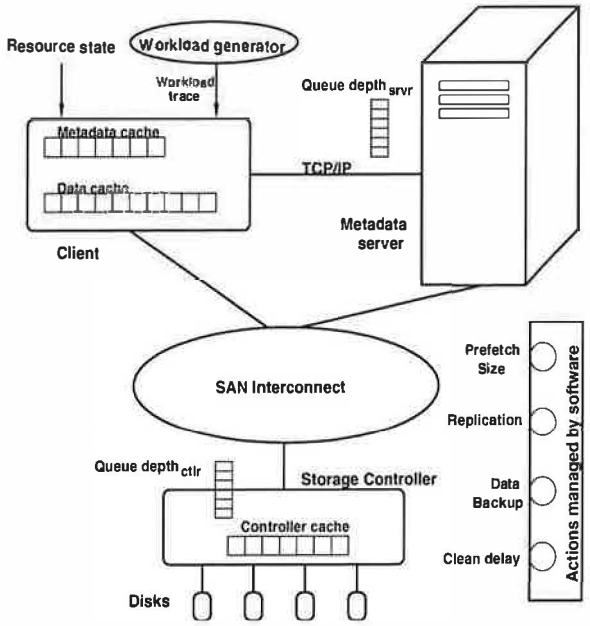


Figure 3. Simulator of a SAN file-system

In this paper, the effectiveness of the competing ECA and Polus management approaches will be discussed using the example of a storage area network (SAN) file system [12, 21]. As shown in figure 3, in a SAN file system, the clients contact a metadata server to obtain the necessary metadata for a particular file. Subsequently the clients go directly to the storage controllers via a SAN protocol to access the storage. The clients cache both the file metadata information and the user block data in two separate caches. In order to write ECA rules for this system, a system administrator needs to do the following:

Establish Goals: System administrators are usually interested in ensuring that certain performance (throughput, latency), reliability and security goals are being met in their SAN file system deployments. For example, they could specify their QoS goals as: (1) ensure that each client has a throughput of at least 40MBps; and (2) ensure that the system has 99.999 percent availability.

Determine the observables to analyze: System administrators have access to many static and dynamic system observables such as the available memory size, the SAN bandwidth being provided to a particular client, and the cache hit rate at the client, the metadata server or the storage controller. The system

administrators also have to access to workload characteristics such as the read/write ratio, workload type (random or sequential), and the block size.

Assess the available actions: System administrators need to be aware of the different possible storage actions that they can perform to manage the storage, such as replication, migration, clean delay [18], request throttling, zoning etc.

Determine thresholds for the observables: Based on prior empirical data or experience, system administrators need to determine the threshold values which, when violated should result in the triggering of corrective management actions. For example, if $cache\_miss\_rate > 20\%$ then take a corrective action.

Select a particular action: If the threshold value of a particular observable is being violated then the system administrator needs to choose a correction action such as increasing the prefetch size or replication of data.

Determine the granularity of the action: For example, when the corrective action being taken is to increase the data prefetch size, then the system administrator needs to also specify the unit of the prefetch size increase.

To put it all together, if the QoS goal of 10 millisecond latency is not being met for a particular client, then the system administrator needs to write the following sets of ECA rules (not exhaustive) to find a remedy:

[Rule 1] If the throughput of a storage controller is at its maximum, then migrate this client's data to another controller that has the necessary available bandwidth.

[Rule 2] If the *client cache miss rate* $> 20\%$ and the workload is sequential then increase the data prefetch size by 4 objects.

[Rule 3] If a particular client is exceeding its allotted bandwidth then throttle its request.

Thus, for a particular QoS goal, the system administrator needs to evaluate the values of all the relevant system observables, assess whether they are violating a particular predetermined threshold value, and then choose a corrective action from a list of possible system management actions. The objective of Polus is to reduce the number of details that a system administrator needs to consider.

3 Details of the Polus Approach

This section presents the details of the Polus framework by both presenting the technical details of the framework and also by illustrating how Polus provides storage management guidance to a Storage Area Network (SAN) file system.

3.1 Polus Terminology

Before describing the system model, we define the terms *behavior, goals and actions.* In Polus, behavior is a defined as a set of QoS dimensions such as $\{throughput, latency, availability, reliability\}$. These dimensions are also referred to as observables. Goals are defined as threshold values on behavior dimensions. e.g. response-time for reads should less than 5 sec, a client's average throughput should be 150 MBps etc. The definition of actions is domain-specific. In Polus, the actions are divided into two categories: a) tunable parameters i.e., changing the value of configuration parameters such a prefetch size, clean-delay, etc., and b) internal actions such as migration, replication, and back-up of data.

The model of the system is shown in figure 4. It consists of two key entities: the management software and the managed system. The management software is assigned QoS goals and is responsible for ensuring that the managed system meets these goals. The interaction between the management software and the managed system is via *sensors* and *actuators.* Monitors gather information about the managed system, while the actuators effect the actions invoked by the management software on the managed system.

The managed system consists of physical *components* that interact to service the application requests. The components service the requests in a particular sequence, with each component processing the information before handing it to the next component. For example, within a SAN file system, the components are client machines, metadata servers, storage controllers and disks. A component has pre-defined properties such as the maximum number of requests that it can service, the error rate, the average down-time etc. Each component consists of one of more atomic entities referred to as *resources.* In other words, resources serve as an abstraction to refer to components in a generic fashion. In Polus, the possible resources are memory, CPU, network, and storage. For example, the storage controller component consists of memory and storage resources, the client machine component consists of CPU, memory and network resource.

Sensors collects information about the *state* of the managed system. The state of the system is defined as a quadruple $<S, R_{current}, IP, B_{current}>$, defined as follows:

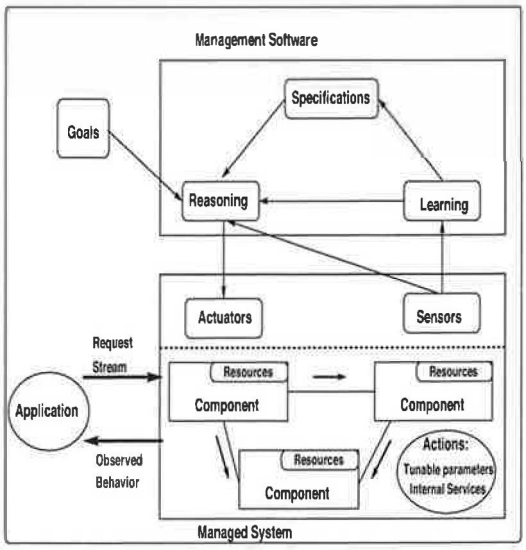


Figure 4. System model

- *Workload Characteristics* (S): S is represented as set of measurable parameters that characterize the application requests. These characteristics are dynamic and constantly changing. For example, in a storage system, typical dimensions are the read/write ratio of access requests, the access pattern (sequential/random), and block size of requests.
- $R_{current}$: Represents the current usage of the resources on a per component basis.
- *Invocation Path* (IP): Represents the sequence in which the components are invoked while servicing the application requests. For example, client machine $\rightarrow$ storage controller 2 $\rightarrow$ LUN 8 represents a possible invocation path
- $B_{current}$: Represents the current values of the behavior dimensions.

Actuators invoke the *actions* selected by the management software. The impact of invoking an action is not constant and is a function of the *state*. Furthermore, whether an action can be invoked or not is dependent on the *state* (i.e. not all actions are applicable within a particular *state*).

3.2 Polus Framework

The three main parts of the Polus framework are:

Modeling of information: The model describes how actions such as prefetching, replication, etc are represented in the system.

Generation of the knowledge base: How rule-of-thumb specifications and learning are used in conjunction to generate the details of the information model.

Reasoning: During the detection of a QoS violation, the reasoning engine decides which action to invoke using the information in the knowledge base. The management semantics of the reasoning engine are expressed using first-order predicate calculus.

3.2.1 Modeling of Information

In Polus, an action is represented as a software object and is referred to as an *action object*. The attributes of the action object are a triplet of the form $< I, P, B >$, defined as follows:

Implications I : This defines the impact of action invocation on the system behavior. The value of the impact function is dependent on the current state.

Preconditions P : This represents the dependencies of the action on system state (i.e. the prerequisites for invoking the action). The prerequisites are defined in terms of thresholds on resources and workload characteristics. Preconditions can be visualized as defining boundaries to the state-space, since the impact of invoking the action beyond the boundaries is not captured by the action object. Preconditions are represented as exclusion/inclusion lists.

Base behavior B : This attribute defines the details associated with the actual action invocation. This includes parameters that need to be passed during invocation function(e.g., to invoke prefetching, the prefetch size needs to be passed), the increment size, and the transient resource costs for action invocation. The details of action invocation are defined in terms of unit invocation, which is similar to that used in implications.

Figure 5 gives the example of the prefetch action object. More complex actions such as replication have additional base invocation parameters such as number of replicas, the data-set to replicate, selecting the component where the replicas will be stored. Polus assumes that the semantics for generating the values for these replication parameters will come from a separate resource planning tool.

3.2.2 Generation of Knowledge base

In Polus, the details of the action object are generated using a combination of rule-of-thumb specifications and learning. Figure 6 shows how the prefetch object is internally maintained by Polus. The details of the action object may not all be available when the system starts

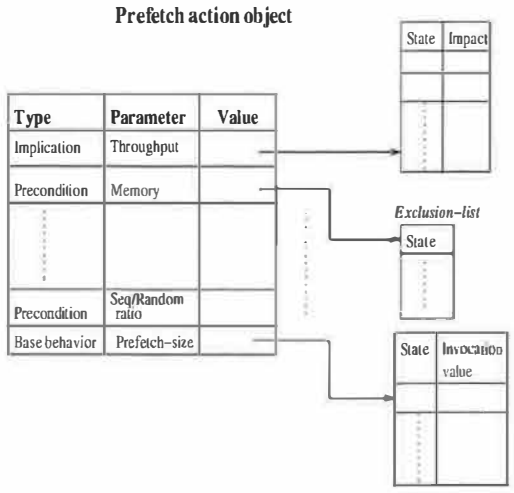


Figure 5. Example of a Prefetch action object

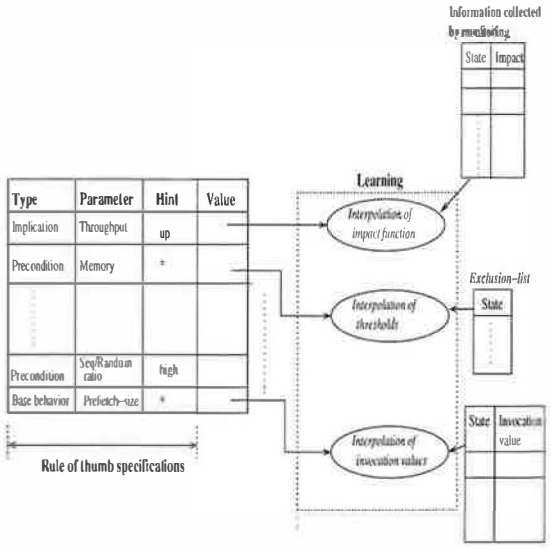


Figure 6. Prefetch action object internally maintained by Polus

off. A part of the information is generated using user-defined specifications (rule-of-thumb) and rest is generated using learning functions (as will be explained in this subsection). There is a "confidence" value associated with every piece of generated information. The confidence value is based on the error value of the learning algorithm that signifies the difference between the predicted value and the actual observed value.

Rule of thumb specifications

The rule of thumb specifications are simple declarative statements. They fundermentally serve two purposes:

- They specify the relationship between actions, resources, workload characteristics, and behavior dimensions. For example, prefetching requires memory.
- They specify "hints" for the possible values associated with the relationships. For example, prefetching is dependent on the sequential/random ratio. A higher value of this ratio is more favorable. This hint helps the learning algorithm to perform linear classification within the behavior space (i.e., if invoking prefetching did not improve throughput when the sequential/random ratio $= \beta$, then the hint allows the learning algorithm to interpolate that for all values less than or equal to β, invoking prefetching may not be beneficial).

The template for the rule-of-thumb specifications has similar categories as those of the action objects:

```
<action name = PREFETCH>
  <behavior_implications>
     <implication dimension = throughput
           impact = up>
  </behavior_implications>
  <preconditions>
     <precond dimension = sequential/random
           ratio value = high>
     <precond dimension = read/write ratio
           value = high>
     <precond dimension = memory value = *>
  </preconditions>
   <base invocation>
     <function name = change_prefetch_size>
     <parameter type = float>
   </base invocation>
</action>
```

Learning

Learning algorithms quantify the rule of thumb specifications and they interpolate the value sets for the relationships defined in the specifications. A learning algorithm is treated as a black box that interpolates information for the $(n+1)^{th}$ data point given a previous sample of n data points. The rule-of-thumb specifications help in pruning the learning space. For example, the implication of invoking prefetching is a function of all the observables, i.e., $Implication(Prefetching) \dashrightarrow f(all\ observables)$. Using the rule of thumb specifications, the interpolation of the implication function is: $Implication(Prefetching) \rightarrow f(throughput, latency)$.

Given that specifications prune the learning space, the question of what happens if the rule-of-thumb specifications are incomplete arises. The current implementation of Polus does not handle this scenario as it assumes that the specifications are complete. However, one can overcome incomplete specifications using existing machine learning approaches such as "bagging" [5], which discover unspecified relationships and add them to the specifications.

In Polus, the process of learning is a combination of off-line training and on-line refinement. Initially, when the management software is installed, learning is an off-line process, which means that the learning algorithms are just recording the system state along-with the actions invoked by the administrator. After a sufficient number of training data points are recorded, the learning algorithm switches to the on-line approach in which it keeps refining the interpolation function generated using the training data points. This refinement is based on the difference between the interpolated value and the value actually obtained from the invocation (also referred to as re-enforcement learning [20]).

Conjunction of specifications and learning

The attributes of the action object are derived using a combination of specifications and learning. In the current implementation of Polus, the rule-of-thumb specifications forms the static part of the knowledge base i.e., the system does not discover additional relationships. The information associated with the learning algorithms forms the dynamic part of the knowledge base as this information is constantly updated by monitoring the system and refining the interpolation functions.

The reasoning engine accesses the information in the knowledge base. The access is a query of the form *Does Action X affect throughput and if yes, by how much?*. The query is translated into a sub-query which is first handled by the specification sub-part of the knowledge base that looks-up to see if there is a relation defined between Action X and throughput. If yes, it generates another query for the learning sub-part that contains the interpolation function for action X, current-state and throughput. Figure 7 illustrates this in the context of prefetching.

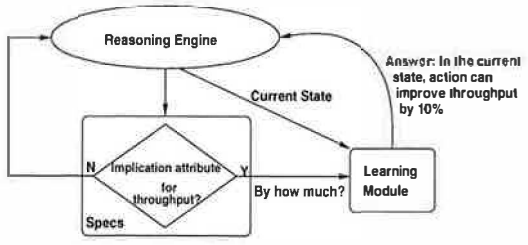


Figure 7. Conjunction of knowledge with the specifications and learning module

3.2.3 Reasoning

The reasoning engine is triggered whenever any of the assigned goals are violated. The objective of the reasoning engine is to select the action(s) that need to be invoked at run-time, in response to the violation of goals. The operation of the reasoning engine is a three step process:

Problem determination: Analyzes the components in the invocation path; determines components that are saturated or the components that need to be tuned for the current state.

Base-level reasoning: Searches the action objects, based on the queries generated by the problem determination module (can be visualized as an object interpreter).

Meta-level reasoning: The meta-level is responsible for higher-order optimizations such as deciding between multiple candidate actions based on invocation cost, error functions, side-effects and so on.

In the SAN file system example, if the prescribed throughput goal of 100 MBps is being violated, the steps that are taken by the reasoning engine to rectify the problem are described in the following subsections.

Problem determination

Input: The current state and the goals being violated.
Output: The components whose behavior needs to change and the type of change. This is expressed as a query of the form:

$$\phi = \{(c, b, change) |\ c \epsilon Component,\ b \epsilon Behavior,\ b_{new} = (1 + change) b_{current}\}$$

Approach: Problem determination has been an area of ongoing research. We briefly describe a simplified approach to illustrate how problem determination, base-reasoning and meta-reasoning work together.

- Determine the components being used in the invocation path.
- Compare the current behavior of these components with the static capabilities of the components. This is similar to system diagnosis using model-based reasoning [22].
- Additionally, compare the current state with a previous state (in which the goals were met). This change analysis generates additional parameters used to search the specifications and tune components in the invocation path that are not saturated. The changes in the system state can be along one or more of the following dimensions: a) Resources utilization (accounts for failures, resource-additions, and application request rate), b) Workload characteristics c) Assigned goals, d) Invocation path (accounts for changes in active datasets or physical components).

In the SAN file system example, by analyzing the invocation path (i.e. client machine, controller, disks), we determine that disks are saturated (i.e. the current I/O rate is the maximum they can support). Furthermore, a change analysis with a previous state reveals that client I/O request rate has increased by 40% and that the sequential/random ratio of the workload has changed from 0.1 to 0.7. Based on the problem determination analysis the following two queries get generated: Query 1: Select an action that improves the throughput of the disks by 25% (the fact that it is saturated will show-up in the preconditions of actions). Query 2: Select an action that improves the throughput of the (controller or client machine) by 25%, and is optimized for sequential workloads.

Base-level reasoning

Input: The set of queries ϕ derived by problem determination
Output: A set of candidate_actions that partially or completely satisfy elements in ϕ.
Approach:
The logic for searching the knowledge base is expressed in first-order predicate calculus. The logic captures the thought process that is implicit while writing imperative specifications. The information model of the action object makes it possible to express these semantics and derive the actions to be invoked "on-the-fly." A few examples of the thought process, expressed as first-order predicates, are described below.

At the high-level, a candidate action is one that affects the component in ϕ, satisfies the preconditions (p) in the current-state, and has the desired implications (i).

$$\forall\, a, s, \phi\ candidate(a, s, \phi) \Rightarrow$$
$$component(a, \phi) \wedge p(a, s) \wedge i(a, s, \phi)$$

where: $\{\, a \in Action,\ s \in State\}$

Satisfy implications (i) is derived (as explained in Section 3.2.2) by combining the implication attribute present in the specifications ($Spec_I$) and the associated interpolation function ($interpolate$) derived by learning.

$$\forall\, a, s, \phi\ i(a, s, \phi) \Rightarrow$$
$$\forall\, x\ Spec_I(a, x) \wedge (interpolate(x, s) > 0)$$

Similarly, satisfy preconditions (p) is defined as:

$$\forall\, a, s\ p(a, s) \Rightarrow \forall\, y\ Spec_P(a, y)$$
$$\wedge\ \neg(Exclusion\_set(y) \in Value(s))$$

At each step in the reasoning process, while selecting the candidate-actions, Polus maintains a log of the available choices and the option that was selected. This can later serve as an explanation to the human administrator, regarding how a specific action was selected.

In the example, assume the two candidate actions that are selected based on Query 1 and 2 are:

1. Invoking prefetching at the client machine
2. Invoking data replication at the disks

Meta-level operations

Input: A set of candidate actions, and ϕ generated by the problem determination module
Output: The actual set of actions to be invoked
Approach:
The aim is to select a candidate action, using an optimization function based on the following parameters:

- The transient overhead associated with the action invocation. For example, invoking replication has more overhead compared to invoking prefetching.
- The behavior side-effects of the action (i.e., an action, in addition to improving the violated goals, can possibly have a negative impact on other behavior dimensions). For example, invoking data backup improves availability, but it has a negative side-effect on throughput and latency.
- The confidence level associated with the details of the action object. As mentioned earlier, the confidence level is associated with the learning function error, measured as a difference between the predicted and the actual observed values.

An important requirement for this optimization is that it should not be based on short term goals (e.g., invoking replication has high overheads, but it might be beneficial in the long-run compared to invoking prefetching multiple times).

In Polus, the optimization algorithm is using n-step look ahead [20], which is extensively used in game theory. In n-step look ahead, the system simulates the impact of pursuing different options. The simulation is repeated n times, and the end result of the simulation is used to decide on the option to be selected in the current state. The simulation is based on the implication and precondition information of the action object. The n-step look ahead is just a rough estimate for the optimization, because external factors such as changes in workload or resources may render the predictions inaccurate. But it definitely helps in detecting instabilities arising due to cycles in state transitions (i.e. the system ping-pongs between two states, invoking the same set of actions repeatedly). It also helps in avoiding choices where a single action leads to a series of actions being invoked due to the cumulative side effects of actions.

Finally, in addition to considering the candidate actions independently, it is possible to reason with combinations of actions. For example, instead of considering the choices of invoking either prefetching or replication, it is possible to consider a combination of prefetching and replication as an additional candidate choice. Composite actions can be handled using vector arithmetic addition techniques but the description of these techniques is beyond the scope of this paper.

4 Experimental Setup

In the experimental setup, a SAN file system simulator is being used as the managed system. For the management software, we use an implementation of the Polus toolkit and compare it with its rule-based ECA counterpart. The Polus toolkit is built using ABLE (Agent Building and Learning Environment) [4]. ABLE provides the basic building blocks for Polus, namely learning algorithms such as neural networks, self-organizing map, JDBC connectivity for interfacing with the database, and data filters. The Polus modules are implemented as Java beans or agents. The implementation consist of agents for: Specifications (input), Reasoning, Learning, Sensors and Actuators. The rest of the section describes the implementation of the file system simulator.

The entities within the SAN file-system simulator are similar to those introduced in Section 2. The cost of atomic operations used in the simulator are shown in table 1. The simulator models the following actions that are invoked by the management software via actuators: Pre-fetch size tuning, Data replication, Backup and Clean-delay interval. I/O operations within the SAN file system are invoked by the client and can have multiple possible paths, depending on whether the data is cache or not. The simulator considers the following paths:

MHDH Metadata and data hit in the client

MHDM Metadata hit and data miss in client

MMDM Metadata and data miss in the client

The summation of each of these probabilities of the invocation paths $P_{MHDH} + P_{MHDM} + P_{MMDM} = 1$

The average I/O latency is given by:

$$L = P_{MHDH} * L_{MHDH} + P_{MHDM} * L_{MHDM} + P_{MMDM} * L_{MMDM}$$

where:

$$L_{MHDH} = L_m + L_d$$
$$L_{MHDM} = L_m + L_{DM}$$
$$L_{DM} = P_{Controller-hit} * Q_C * S_{Controller} + (1 - P_{Controller-hit}) * Q_C * S_{Disk}$$
$$L_{MMDM} = Queue\_depth_{Server} * S_{Server} + L_{DM}$$

Operation	Cost
Size of metadata object	1000 bytes
Latency to access from metadata cache (L_m)	20 μsec
Latency to read from datacache (L_d)	20 μsec
Service time of server (S_{server})	420 μsec
Service time from controller cache ($S_{controller}$)	0.6 $msec$
Service time of disk (S_{disk})	6 $msec$
Queue depth at controller (Q_C)	256 (max)
Size of metadata cache	128 MB
Size of data cache	1 GB
Size of controller cache	256 MB

Table 1. Cost of atomic operations in the file system simulator

The values for probabilities such as P_{MHDH}, and $P_{Controller-hit}$ are modeled by representing the caches as finite sized-arrays and keeping track of data blocks in the elements. Similarly, the average queue depth such as Q_C are actually modeled by using service time to complete each request.

Each action is modeled to reflect its impact on the invocation path, system resources and changes in the workload characteristics (table 2). To activate the actions in the file system simulator, specifications are fed into Polus and rule-based management. The specifications for a rule-based system consist of ECAs that describe the system-behavior for different system-states. The ECA specifications in this example run into 7 pages (76 rules). The exact Polus specifications that are fed are given in figure 8.

5 Experimental Analysis

The experimental analysis consists of a quantitative comparison of Polus and Rule-based systems for different system states. During evaluation, the values (thresholds and action invocation) for rule-based systems are assumed to be correct and empirically obtained from prior runs.

The file system is driven by a trace generator that imposes different states on the system. The generated

Action	Description	Invocation path parameters affected	Resources affected
Prefetching	Readahead of data and metadata	P_{MHDH} and P_{MHDM}	Data cache, metadata cache and interconnect bandwidth
Replication	Creates replica of data on a different volume in the controller	Q_C	Storage space (ignoring transient effects)
Data backup	Consider only transient effects since we are not considering availability	P_{MHDH}, P_{MHDM}, P_{MMDM} Q_C, Q_{Server}	Memory, interconnect bandwidth and storage space
Clean delay	Frequency at which dirty buffers are flushed to disks	Only for writes – P_{MHDH} bursty traffic for storage controller Q_C:	Metadata cache and data cache (metadata cannot be evicted till data is written)

Table 2. Modeling actions within the file system simulator

Action	Implications	Preconditions	Base Invocation
Prefetch	<Throughput, impact = up>	<Workload = sequential/random, value = high> <precond dimension = memory, value >20%> <precond dimension = fc_bandwidth, value = *>	*Parameter:* prefetchSize *Function:* changePrefetchSize
Replication	<Availability, impact = up> <Latency, impact = up> <Throughput, impact = depends>	<Workload = read/write, value = high> <Workload = Queue-depth, value > 16> <Resource = storage-disks, value = *>	*Parameter:* numReplicas *Function:* invokeReplication
Clean delay	<Latency, impact = up> <Throughput, impact = depends> <Reliability, impact = down>	<Workload = read/write, value = low> <Workload = writes, value = async> <Resource = memory, value = *>	*Parameter:* cleanDelay interval *Function:* changeDelay
Data backup	<Latency, impact = down> <Throughput, impact = down> <Reliability, impact = up>	<Trigger = backup, value = *> <Resource = storage value > 35%>	*Parameter:* backupThroughput *Function:* invokeBackup

Figure 8. Specifications fed to the Polus framework

state is a triplet of the form: <Workload characteristics, Available Resources, Goals >. The values for the goals are different than their current values. For each of these system states, we compare the response of both Polus and an ECA based rule system. Table 3 categorizes the possible system state.

The analysis of Polus and ECA for each of the categories is described as follows:

Category 1: Single action applicable

Analysis: The comparison is shown in figure 9. This is a simple category with a single candidate action. Polus generally selects the same action as an ECA-based system. ECA is assumed to have the right value for invocation while Polus uses the incremental approach for invocation. Learning improves the incremental approach by interpolating the starting point for incremental invocation.

Insights: The efficiency of the incremental algorithm is dependent on the impact function of the invoked action,

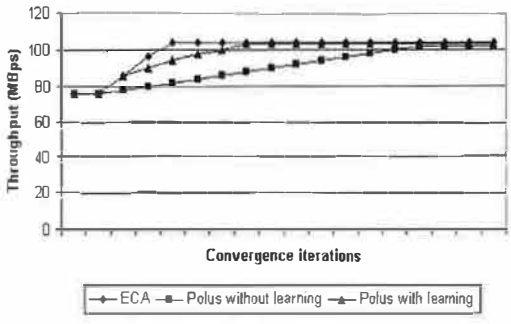


Figure 9. Comparing Polus and ECA for category 1 (single candidate action). In the graph, the throughput goal = 100 MBps.

which could be linear, quadratic, exponential and so on.

Category 2: Multiple actions applicable

Analysis: This category (figure 10) exposes the "weak-spot" in Polus. When the candidate actions are indis-

Categories	Description	Example (File system simulator)
Category1: Single action applicable	The system states in this category are such that only a single candidate action is applicable, i.e., searching the specifications leads to a single candidate action	Workload: Sequential, read dominated with read/write ratio of 0.9, avg. queue-depth = 6 Current Throughput = 80 MBps Goal = 100 MBps Polus specification search: The only action that becomes applicable is Prefetching.
Category 2: Multiple actions applicable (they appear to have similar preconditions and/or implications)	In this category more than one action have similar preconditions and/or implications and are indistinguishable. In reality, these actions are not similar This category becomes increasingly common during initial bootstrapping, i.e., the system hasn't learnt values for preconditions and implications	Workload: Read dominated, sequential/random ratio = 0.2, average queue-depth = 8. Current Throughput = 80 MBps Goal = 100 MBps Polus specification search: Prefetching and Replication are selected as candidate actions. In reality Prefetching is not applicable as the workload is not sequential, but Polus does not have the threshold value for the sequential/random ratio in prefetch specifications
Category 3: More than one goal not met	In this category, more than one action needs to be invoked as a single action cannot satisfy the goal requirements	Workload: Sequential, read/write= 0.3 Current Throughput = 80 MBps Goal = 100 MBps , Current Latency = 6msec Goal = 4.5 msec Polus specification search: Prefetching and Clean delay are both invoked as the former improves throughput while the later improves latency
Category 4: Recurrent action invocation (One action, leads to chaininvocation of actions)	In this category, invocation of an action leads to a chain-invocation of a series of actions. Ability to detect and prevent recurrent action invocation is a required property of the management software	Workload: Trigger for data backup with window = 4 hours Polus specification search: Theoretically, backup can be invoked since the goals are being met. But invoking backup at this time will cause latency goals to be overshot
Category 5: No action applicable (Negation of previous actions required)	Actions are negated under two scenarios: to make resources available for another actions, and the workload preconditions change	Workload: Changes from large block sequential to small block random Polus specification search: Prefetching hurts performance as memory and storage resources are used for acquiring data that is never used

Table 3. Categorizing the possible system states

tinguishable, Polus tries them one-by-one till it either leads to a negative impact on the observable values or the goals are met. As shown in the graph, Polus initially selects the wrong action (i.e. prefetching). After the value dips further, Polus tries the next candidate action (i.e. replication). Learning adds the threshold values (in this example at the pre-conditions level) and enables distinguishing between the actions.

Insights: In systems with larger action-sets, it is quite possible that Polus never converges due to side-effects of trying wrong actions.

Category 3: More than one goal not met

Analysis: Rule-based systems will invoke a single action in each iteration without analyzing the combined impact of the actions. On the other hand, Polus considers different permutations to combine the actions (figure 11). This is beneficial when the two actions act on the same resources, such that the invocation of one action beyond a threshold can violate the pre-conditions of other actions. As shown in the graph, ECA does not meet the latency goal due to lack of memory resources. In its previous iteration Prefetching was invoked for throughput goals and the rules did not consider the combined state while deciding the value for prefetching. Learning refines the attributes of actions allowing better combination strategies.

Insights: Higher-order operation are powerful in deriving permutations that cannot be possibly defined statically.

Category 4: Recurrent action invocation

Analysis: The comparison in shown in figure 12. Polus uses look ahead while invoking actions to estimate the impact of the action on the goals. Rule-based systems don't have an equivalent of this (though it is possible

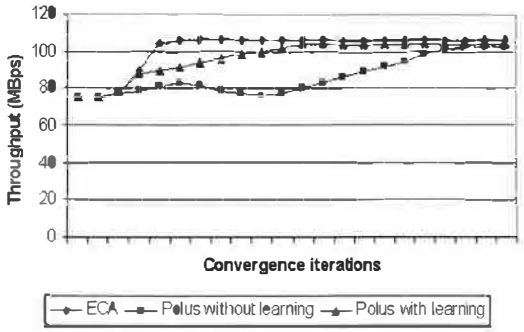


Figure 10. Comparing Polus and ECA for category 2 (multiple candidate actions). In the graph, the throughput goal = 100 MBps

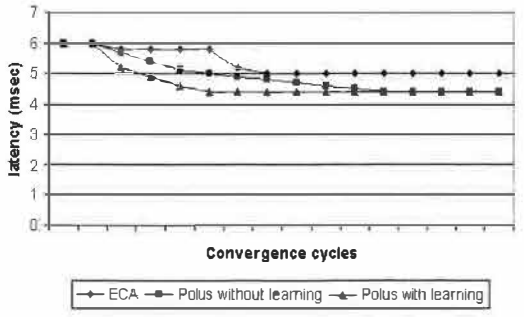


Figure 11. Comparing Polus and ECA for category 3 (more than one goal not met). In the graph, the latency goal = 4.5 msec

to write separate rules to cater for this). As shown in the graph, ECA invokes the Backup action that leads to invocation of a series of actions (Replication in this example). Polus does a look-ahead and does not invoke Back-up during the current system-state. (For Back-up we are assuming a time-window)

Insights: Look-ahead is a required operation and effective only with some learning of the action model. Hence there is only a single curve in the graph.

Category 5: Negation of previous actions

Analysis: Both Polus and Rule-based systems can cater to negation of actions (figure 13). ECA can accommodate by writing separate rules. Learning does not play a significant role in this category

Insights: Explicit need to write separate rules in rule-based systems for negation whereas the Polus reasoning engine can account for this without any additional specifications.

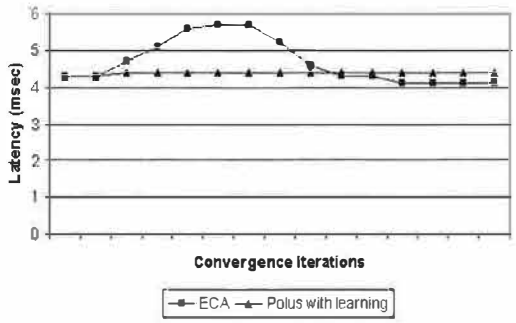


Figure 12. Comparing Polus and ECA for category 4 (recurrent action invocation). In the graph, the latency goal = 4.5 msec

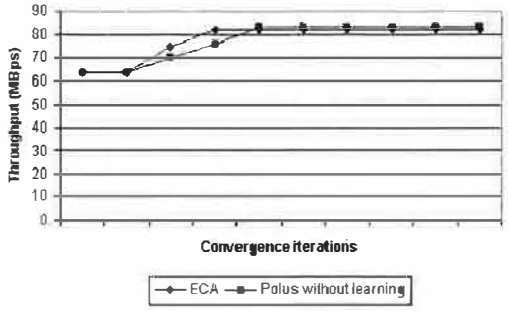


Figure 13. Comparing Polus and ECA for category 5 (negation of previous actions required). In the graph, the throughput goal = 80 MBps

6 Discussion

During the evaluation of the Polus framework, we made a few simplifying assumptions.

- The system state is assumed to remain constant during the time that an action is invoked. While it may be argued that this is an unrealistic assumption for actions that require significant time to complete, it is a reasonable one for most actions that involve small overhead.
- We do not consider the case where the specification provided by a administrator may be incomplete or contains incorrect information that may mislead the management software in its decision making.
- The experimental evaluation contains a single implementation of learning, reasoning and base invocation algorithms. These algorithms have not been fine tuned or optimized, and it is possible to plug-in more sophisticated algorithms for tree searching, incremental base invocation and learning. An evaluation by varying these algorithms is beyond the scope of this paper.

In real life deployments of Polus, system vendors can provide templates containing rules of thumb specifications and initial threshold values (obtained from training runs) for different workloads and system configurations. The adaptive behavior of the Polus framework will fine tune the knowledge base (i.e. threshold values) and tailor it to specific user environment and workload characteristics.

The evaluation framework presented in the paper is a system with just four possible actions. Hence, it is important to note that our aim was to understand the possible weaknesses of the Polus approach, and to get a first cut estimate of the number of iterations required in converging towards a specified QoS goal for different system states. As shown in the experiment section, the initial results show that this approach is promising; thus, we are currently implementing Polus as part of a real storage QoS management system. Polus can also be initially deployed as an aid to system administrators that allows them to perform what if analysis with respect to whether a system can support different QoS goals.

7 Related Work

Rome [25], Minerva [1], Hippodrome [2], and "attribute-managed storage" [13] projects from HP, SELF* project [9] from Carnegie Mellon, Storage Tank [21], SledRunner [6] projects from IBM, Control Centre product line from EMC, Storage Central product line from Veritas, and BrightStor product line from Computer Associates all aim to simplify storage management by automating different aspects of storage management. The Polus framework presented in this paper is complementary to these projects, since none of these projects specifically address the QoS goal transformation problem being addressed in this paper. Moreover, there is nothing inherent in the Polus framework that prevents its adoption by these different frameworks and products as part of their QoS solutions.

The Polus framework was built using specification, learning and reasoning techniques from the artificial intelligence (AI) domain. These technologies have a proven track record as they have been successfully used to build expert systems in medical, system configuration, video games and speech/handwriting processing application domains. To the best of our knowledge, Polus is the only system of its kind (in the domain of storage performance management) that integrates a rules-of-thumb specification model, reasoning (including higher-order operations) and a self-refining learning engine to manage a storage system.

Polus leverages concepts in AI and uses them as building blocks in its solution. Techniques for specification in expert systems are broadly classified as imperative (e.g. rule-based), declarative (e.g. logic programming) or mixed. Brittleness has been identified as the biggest drawback of imperative rule-based systems [7], whereas, logic based systems overcome this problem by using a reasoning engine to combine facts/beliefs in the knowledge base to draw conclusions. The Polus specification of action attributes is similar to the declarative approach. Further, reasoning in Polus is a combination of specification search algorithms and higher-order operations. Polus uses forward chaining to search the specifications, but it is possible to use other approaches such as backward chaining or heuristic-based searching. Other popular approaches for reasoning are: Model-based, Constraint-based, and Case-based reasoning [17]. As explained earlier, Polus uses CBR as part of the reasoning engine to tie in the knowledge acquired by learning in the decision-making. Finally, learning in Polus systematically refines the specifications. It leverages research in the domain of machine learning algorithms such as neural networks and reinforcement learning [11, 15].

Currently, there are many competing policy specification standards [14, 19]. Polus can leverage any one these existing standards for specifying the base rules-of-thumb. Furthermore, there are no in-built dependencies that prevent Polus from leveraging the canonical SNIA SMI-S storage device standard [23] as the representation for low level system actions.

A Case-Based Reasoning approach, in which a system starts off with no specifications and uses the previously learnt cases to decide how a goal should be transformed, has been employed in the web-server configuration domain [24]. The bootstrapping behavior of that approach is not attractive in real-world scenarios where the reasonable number of cases that need to be learned a priori are zero (resource states, workload characteristics, goals, action set). In comparison, as shown in the experiment section, the combination of rule-of-thumb specification and a learning engine has a reasonable bootstrapping behavior. That is, the Polus approach is able to dynamically adapt even when it does not start from the most desired bootstrapped state. Mark, et al., [3] propose an approach to separate the goal from the base rule specification. They create a mapping between the rule and user-requirements, making it easy for validation and usage. The Polus approach is more sophisticated, in that it encodes the goal implications and uses them to automate the reasoning process.

Another approach [8] uses genetic algorithms for self-tuning. In this approach each system parameter is tuned by an individual algorithm and the genetic algorithm decides the best combination of algorithms. Unlike Polus, this approach does not allow refinement of the decision-

making based on learning. Zinky, et al., [27] present a general framework, called QuO, to implement QoS-enabled distributed object systems. The QoS adaptation is achieved by having multiple implementations. Each implementation is mapped to an environment and to a QoS region. The QuO approach is static, as it does not implement semantics for reasoning about the various possible configurations.

8 Conclusion

Policy-based QoS management has been advocated as a "silver bullet" that can help to drastically increase the amount of storage that can be managed by system administrators. It is typically implemented using the ECA rules mechanism. However, as shown in this paper, the policy-based QoS management approach is not gaining much traction because of the associated difficulty in mapping high level QoS goals to low level system actions using the ECA approach.

In this paper we provide an alternative paradigm for mapping high level QoS goals to low level system actions. Our Polus approach leverages the proven AI techniques of learning and reasoning, and combines them with a declarative specification approach. Using this approach, system administrators can specify general rules of thumb to describe their knowledge instead of complex ECA rules containing detailed threshold values. In Polus, these threshold values are derived using learning algorithms. Furthermore, the use of reasoning engine allows Polus to automatically select the right action from amongst the various competing alternatives.

In conclusion, this paper presents a new approach towards how QoS goals can be mapped to low level system management actions. The aim of this approach is to reduce the number of inputs that are required from system administrators. We have analyzed the key concepts of this approach by using Polus to manage a simulated SAN file system and this is the first stepping stone towards the use of Polus-like approaches for managing real storage systems.

References

[1] G. Alvarez, E. Borowsky, S. Go, T. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM Trans. Comput. Syst.*, 19(4):483–518, Nov. 2001.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proc. 1st Conf. on Filesystem and Storage Tech.*, pages 175–188, June 2002.

[3] M. Bearden, S. Garg, W. Lee, and A. van Moorsel. User-centric QoS policies, or saying what and how. Work-in-progress report, Proc. 11th Workshop on Distributed Systems: Operations and Management (DSOM), Dec. 2000.

[4] J. Bigus, D. Schlosnagle, J. Pilgrim, W. M. III, and Y. Diao. ABLE: A Toolkit for Building Multiagent Autonomic Systems. *IBM Sys. J.*, 41(3), Sept. 2002.

[5] L. Breiman. Bagging predictors. *Mach. Learning*, 24(2):123–140, 1996.

[6] D. Chambliss, G. Alvarez, P. Pandey, R. M. D. Jadav, J. Xu, and T. Lee. Performance virtualization for large-scale storage systems. In *Proc. Symp. on Reliable Distributed Sys. (SRDS)*, Oct. 2003.

[7] V. Dhar and H. Pople. Rule-based versus structure-based models for explaining and generating expert behavior. *Comm. ACM*, 30(6), June 1987.

[8] D. Feitelson and M. Naaman. Self-tuning systems. *IEEE Software*, 16(2):52–60, 1999.

[9] G. Ganger, J. Strunk, and A. Klosterman. Self-* Storage: Brick-Based Storage with Automated Administration. Technical Report CMU-CS-03-178, Carnegie-Mellon University, Aug. 2003.

[10] M. Genesereth and M. Ginsberg. Logic Programming. *Comm. ACM*, 28(9), Sept. 1985.

[11] J. Ghosh and A. Nag. *An Overview of Radial Basis Function Networks.* Radial Basis Function Neural Network Theory and Applications, Physica-Verlag, 2000.

[12] G. A. Gibson, D. F. Nagle, W. C. II, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka. NASD Scalable Storage Systems. In *Proc. of Extreme Linux Workshop in the 1999 USENIX Ann. Tech. Conf.*, June 1999.

[13] R. Golding, E. Shriver, T. Sullivan, and J. Wilkes. Attribute-managed Storage. In *Proc. Workshop on Modeling and Specification of I/O*, Oct. 1995.

[14] IETF Policy Framework Working Group. IETF Policy Charter. http://www.ietf.org/html.charters/policy-charter.html.

[15] T. Kohonen. *Self-Organizing and Associative Memory 3rd ed.* Springer-Verlag, 1988.

[16] R. Kowalski. Algorithm = logic + control. *Comm. ACM*, 22(7):424–436, 1979.

[17] D. Leake. *Case-Based Reasoning: Experiences, Lessons and Future Directions.* AAAI Press, 1996.

[18] J. Mogul. A better update policy. In *Proc. Summer 1994 USENIX Conf.*, pages 99–111, June 1994.

[19] B. Moore. Network Working Group – RFC3060. Policy Core Information Model – Version 1 Specification. http://www.ietf.org/rfc/rfc3060.txt, 2001.

[20] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann Publishers Inc., 1992.

[21] D. Pease, J.Menon, B. Rees, L. Duyanovich, and B. Hillsber. IBM Storage Tank-A heterogeneous scalable SAN file system. *IBM Sys. J.*, 42(2):250–267, 2003.

[22] A. Singhal, G. Weiss, and J. P. Ros. A model-based reasoning approach to network monitoring. In *Proc. Workshop on Databases*, pages 41–44. ACM Press, 1997.

[23] Storage Networking Industry Association. SMI Specification version 1.0. http://www.snia.org, 2003.

[24] D. Verma and S. Calo. Goal Oriented Policy Determination. In *Proc. 1st Workshop on Algorithms and Architectures for Self-Managing Sys.*, pages 1–6. ACM, June 2003.

[25] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. *Lecture Notes in Computer Science 2092*, pages 75–91, 2001.

[26] J. Wilkes. Data Services - from data to containers. Keynote address, 2nd Conf. on Filesystems and Storage Tech. (FAST), Apr. 2003.

[27] J. A. Zinky, D. E. Bakken, and R. D. Schantz. Architectural support for Quality-of-Service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.

Buttress: A toolkit for flexible and high fidelity I/O benchmarking

Eric Anderson Mahesh Kallahalla Mustafa Uysal Ram Swaminathan

Hewlett-Packard Laboratories
Palo Alto, CA 94304
{anderse,maheshk,uysal,swaram}@hpl.hp.com

Abstract

In benchmarking I/O systems, it is important to generate, accurately, the I/O access pattern that one is intending to generate. However, timing accuracy (issuing I/Os at the desired time) at high I/O rates is difficult to achieve on stock operating systems. We currently lack tools to easily and accurately generate complex I/O workloads on modern storage systems. As a result, we may be introducing substantial errors in observed system metrics when we benchmark I/O systems using inaccurate tools for replaying traces or for producing synthetic workloads with known inter-arrival times.

In this paper, we demonstrate the need for timing accuracy for I/O benchmarking in the context of replaying I/O traces. We also quantitatively characterize the impact of error in issuing I/Os on measured system parameters. For instance, we show that the error in perceived I/O response times can be as much as $+350\%$ or -15% by using naive benchmarking tools that have timing inaccuracies. To address this problem, we present *Buttress*, a portable and flexible toolkit that can generate I/O workloads with microsecond accuracy at the I/O throughputs of high-end enterprise storage arrays. In particular, Buttress can issue I/O requests within $100\propto s$ of the desired issue time even at rates of 10000 I/Os per second (IOPS).

1 Introduction

I/O benchmarking, the process of comparing I/O systems by subjecting them to known workloads, is a widespread practice in the storage industry and serves as the basis for purchasing decisions, performance tuning studies, and marketing campaigns. The main reason for this pursuit is to answer the following question for the storage user: "how does a given storage system perform for my workload?" In general, there are three approaches one might adopt, based on the trade-off between experimental complexity and resemblance to the application:

a) Connect the system to the production/test environment, run the real application, and measure application metrics;

b) Collect traces from a running application and replay them (after possible modifications) back on to the I/O system under test; or

c) Generate synthetic workloads and measure the I/O systems performance for different parameters of the synthetic workload.

The first method is ideal, in that it measures the performance of the system at the point that is most interesting: one where the system is actually going to be used. However, it is also the most difficult to set up in a test environment because of the cost and complexity involved in setting up real applications. Additionally, this approach lacks flexibility: the configuration of the whole system may need to be changed to evaluate the storage system at different load levels or application characteristics.

The other two approaches, replaying traces of the application and using synthetic workloads (e.g., SPC-1 benchmark [9]), though less ideal, are commonly used because of the benefits of lower complexity, lower setup costs, predictable behavior, and better flexibility. Trace replay is particularly attractive as it eliminates the need to understand the application in detail. The main criticism of these approaches is the validity of the abstraction, in the case of synthetic workloads, and the validity of the trace in a modified system, in the case of trace replay.

There are two aspects of benchmarking: a) constructing a workload to approximate a running environment (either an application, trace, or synthetic workload), and b) actually executing the constructed workload to issue I/Os on a target system. This paper focuses on the latter aspect; in particular, we focus on accurately replaying traces and generating synthetic workloads.

The main assumption in using traces and synthetic workloads to benchmark I/O systems is that the workload being generated is really the one that is applied to the test system. However, this is quite difficult to achieve. Our results indicate that naive implementations of benchmarking tools, which rely on the operating system to sched-

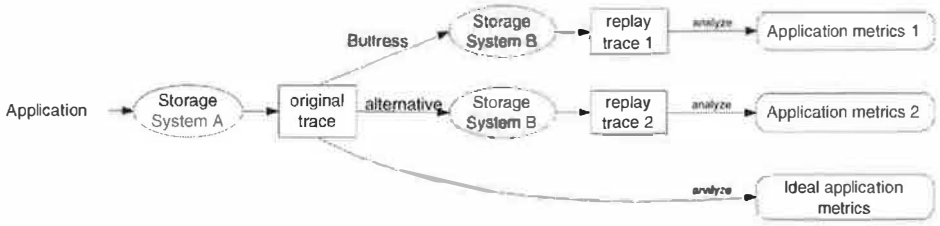


Figure 1: Illustration of our experimental methodology to compare performance of different trace replay techniques. The input is the original trace of an application running on storage system A. We then replay the trace on system B using different trace replay techniques and gather the resulting I/O trace (replay traces). We analyze the resultant traces to determine parameters of interest to the application, such as response times and queue lengths. We then use these metrics to compare the different trace replay techniques among each other and with the original trace if the storage systems A and B were the same.

ule I/Os, could skew the mean inter-I/O issue times by as much as 7ms for low I/O loads. This is especially erroneous in the case of high-end storage systems which might have response times in the 100s of microseconds, and can handle 10s of thousands of I/Os per second. As we shall show in Section 2, this deviation can have significant impact on measured system parameters such as the mean device response time.

The main challenge in building useful benchmarking tools is to be able to generate and issue I/Os with accuracies of about 100μs, and at throughputs achieved by high-end enterprise storage systems. In this paper, a) we quantitatively demonstrate that timing errors in benchmarking tools can significantly impact measured system parameters, and b) we present and address the challenges in building a timing accurate benchmarking tool for high end storage systems.

The rest of the paper is organized as follows. In Section 2 we analyze the impact of not issuing I/Os at the right time on system metrics. Motivated by the need for having accurate benchmarking tools, we first present the complexities in designing such a system which runs on commodity operating systems in Section 3. In Section 4 we present solutions in terms of a flexible and nearly symmetric architecture for Buttress. We detail some specific optimizations of interest in Section 5. Experiments to validate that our implementation achieves high fidelity are described in Sections 6. We conclude with related work in Section 7 and a summary in Section 8.

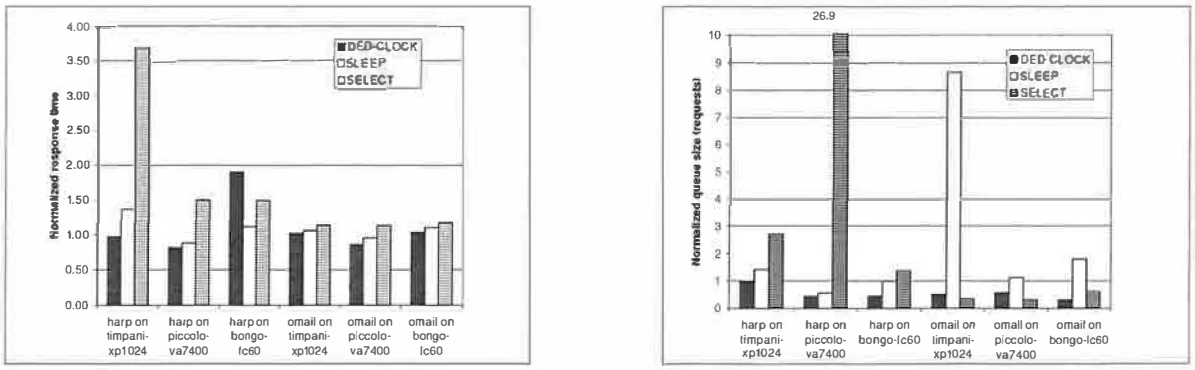
2 Need for high I/O issue accuracy

In this section, we quantify the impact of errors in issuing I/O at the designated time on measured application statistics. We define *issue-error* as the difference in time between when an I/O is intended to be issued and when it is actually issued by the benchmarking tool. One may intuitively feel that I/O benchmarks can adequately characterize applications despite timing inaccuracies in issuing I/O, as long as the remaining characteristics of the workload, such as sequentiality, read/write ratio, and request offsets are preserved. In fact, most studies that do system benchmarking seem to assume that the issue accuracy achieved by using standard system calls is adequate. Our measurements indicate that this is not the case and that errors in issuing I/Os can lead to substantial errors in measurements of I/O statistics such as mean latency and number of outstanding I/Os.

Figure 1 illustrates our evaluation methodology. We use I/O trace replay to evaluate different mechanisms of I/O scheduling, each attempting to issue the I/Os as specified in the *original trace*. The I/O trace contains information on both when I/Os were issued and when the responses arrived. During trace replay, we collect another trace, called the *replay trace*, which includes the responses to the replayed I/Os. We then analyze the traces to get statistics on I/O metrics such as I/O response times and queue lengths at devices. We use these metrics to compare the different trace replay methods.

Note that the I/O behavior of an application depends upon the storage system; hence the I/O trace of the application running on system A is generally quite different from the I/O trace on system B. We expect the I/O issue times to be similar if the replay program is accurate, though the response time statistics and I/O queue sizes on system B are likely to be different. In practice, we rarely have the ability to use the actual application on system B; for rare cases that we could run the application on system B, we collect a replay trace running the application and use it as an ideal baseline. We compare the results of the analysis of the different traces between each other and to the results of the analysis of the ideal trace to evaluate the impact of I/O issue accuracy on the storage system performance metrics.

We used four different mechanisms to replay the application trace (original trace) on the storage system B.



(a) Response time

(b) Queue size

Figure 2: Impact of I/O issue accuracy (normalized to Buttress) on the application I/O behavior on various systems. All the numbers are normalized to the value of the metric reported by Buttress.

All programs we used were multi-threaded and used the pthreads package. We issued I/Os synchronously, one per thread, and all programs used the same mechanism for I/O issue. The most elaborate of these programs is *Buttress* and is the main subject of this paper – we briefly describe the other three programs below.

The first two programs, SELECT and SLEEP used standard OS mechanisms to schedule and issue I/Os (select() and usleep() system calls respectively) to wait until the time for an I/O issue arrives. Each of these had a number of worker threads to issue I/Os and a master thread that hands I/Os to available worker threads. Once a worker thread is assigned to issue an I/O, it sleeps using either the select() or the usleep() call, and the OS scheduler wakes the worker when the time for the I/O arrives. These two programs rely entirely on standard OS mechanisms to keep the time and issue the I/Os and hence their accuracy is determined by the scheduling granularity of the underlying OS.

The third program, DED-CLOCK, uses a dedicated clock thread, and CPU cycle counters to schedule the I/Os. The clock thread spins continuously and wakes up worker threads at the appropriate times and hands them I/Os to issue. The CPU cycle counters are usually much more precise than the standard OS timers, but the throughput of this approach depends on how fast the clock thread can wake up workers to issue I/Os.

These three programs are simple approaches of how one might normally architect a trace-replay program using existing mechanisms. In general, the problem with these approaches is that the accuracy of I/O scheduling is contingent upon the thread being scheduled at the right time by the OS. As a result, they are unable to replay I/O bursts accurately and tend to either cluster I/Os at OS scheduling boundaries or flatten bursts.

Figure 2 presents two storage-level performance metrics using various trace-replay mechanisms. It shows the relative change in I/O response time and average queue size using two applications (omail and harp), across three different storage arrays and benchmark systems (the details of the experimental setup are in Section 6). The figure presents the average measured response time, when different trace replay tools were used, normalized to the average response time when Buttress was used. The main point from these graphs is that the inaccuracies in scheduling I/Os in time may result in as much as a factor of 3.5 difference in measured response time and a factor of 26 in measured queue sizes (both happen for SELECT) – these differences are too large to ignore.

3 Main challenges

It is surprisingly difficult to achieve timing accuracy for low and moderate I/O rates, and even harder for the high rates that enterprise class disk arrays can support. Achieving timing accuracy and high throughput involves coping with three challenges: a) designing for peak performance requirements, b) coping with OS timing inaccuracy, and c) working around unpredictable OS behavior.

First, it is a challenge to design a high performance I/O load generator that can effectively utilize the available CPU resources to generate I/Os at high rates accurately. Existing mid-range and high-end disk arrays have hundreds to thousands of disk drives, which means that a single array can support 100,000 back-end IOPS. The large array caches and the high-speed interconnects used to connect these arrays to the host systems exacerbate this problem: workloads could achieve 500,000 IOPS with cache hits. These I/O rates imply that the I/O work-

load generators have about a few microseconds to produce each I/O to attain these performance rates. Therefore it is necessary to use multiple CPUs in shared memory multiprocessors to generate these heavy workloads.

Second, the scheduling granularity of most operating systems is too large (around 10ms) to be useful in accurately scheduling I/Os. The large scheduling granularity results in quantization of I/O request initiations around the 10ms boundary. This is despite the fact that most computer systems have a clock granularity of a microsecond or less. As shown in Figure 2, this quantization effect distorts the generated I/O patterns, and as a result, the observed behavior from the I/O system with a synthetic load generator does not match the observed behavior under application workload (details in Section 6).

Third, the complexity of modern non-real-time operating systems usually results in unpredictable performance effects due to interrupts, locking, resource contention, and kernel scheduling intricacies. These effects are most pronounced for the shared memory multiprocessor platforms as the OS complexity increases. For example, calling the gettimeofday() function on an SMP from multiple threads may cause locking to preserve clock invariance, even though the threads are running on separate processors. An alternative is to use the CPU cycle counters; however, this is also complicated because these counters are not guaranteed to be synchronized across CPUs and a thread moving from one CPU to another has difficulty keeping track of the wall clock time.

4 Buttress toolkit

Based on our discussion in Sections 2 and 3, and our experience with using I/O benchmarking tools, we believe that a benchmarking tool should meet the following requirements:

a) **High fidelity**: Most I/Os should be issued close (a few μs) to their intended issue time. Notice that a few μs is adequate because it takes approximately that much time for stock OSs to process an I/O after it has been issued to them.

b) **High performance**: The maximum throughput possible should be close to the maximum achievable by specialized tools. For instance, in issuing I/Os as-fast-as-possible (AFAP), the tool should achieve similar rates as tools designed specifically for issuing AFAP I/Os.

c) **Flexibility**: The tool should be able to replay I/O traces as well as generate synthetic I/O patterns. It should be easy to add routines that generate new kinds of I/O patterns.

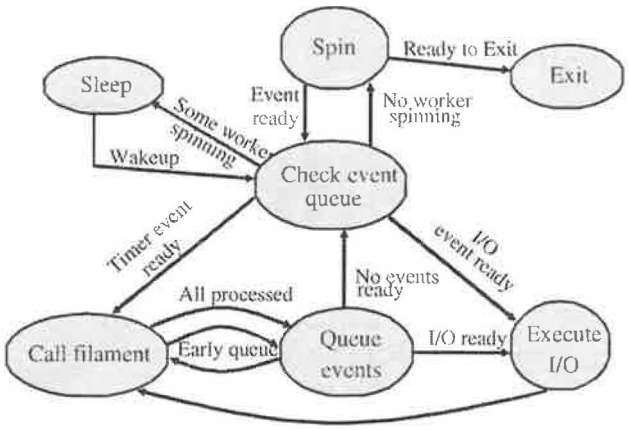


Figure 3: Worker thread state transition diagram in Buttress. The nearly symmetric architecture (w.r.t workers) means that all workers use the same state transition diagram except a low priority thread spinning for timeout.

d) **Portability**: To be useful the tool should be highly portable. Specifically it is desirable that the tool not require kernel modification to run.

In the rest of this section and Section 5, we describe how we developed Buttress to satisfy these requirements. In Buttress we architecturally separated the logic for describing the I/O access pattern and the functionality for scheduling and executing I/Os. This separation enables Buttress to generate a variety of I/O patterns easily. Most of the complexity of Buttress is in the "core", which is responsible for actually scheduling and executing I/Os. The Buttress core is architected as a multi-threaded event processing system. The individual threads are responsible for issuing the I/Os at the right time, and executing the appropriate I/O generation function to get future I/Os to issue.

As implemented currently, Buttress does not require any kernel modifications. It uses POSIX pthreads and synchronization libraries to implement its threads and locking. This makes Buttress very portable – we have been running Buttress on both Linux and HPUX. On the flip side, the performance of Buttress in terms of its maximum throughput and accuracy in issuing I/Os depends on the performance of the underlying OS.

4.1 Filaments, event, and workers

The logic for determining the I/O access pattern is implemented in a collection of C++ objects, called *filaments*. The functionality for scheduling and executing I/Os is embedded in threads called *workers*. The implementation of the workers and the interface to filaments forms the core

of Buttress. Filaments themselves are written by Buttress' users, and currently we provide a library of filaments to generate common I/O patterns.

A filament is called with the *event* that triggered a worker to call that filament. The filament then generates additional events to occur in the future, and queues them up. Workers then remove events from the queues at the time the event is to occur, and process them by either calling the appropriate filament at the right time, or issuing the I/O if the event represents an I/O. Currently, we have three types of events in Buttress:

a) *Timer events* are used to schedule callbacks to filaments at appropriate times;
b) *I/O events* are used to schedule I/O operations. The event object encapsulates all the information necessary to execute that I/O. The I/O completion events are used by workers to indicate I/O completion to filaments; and
c) *Messaging events* are used to schedule an inter-filament message to be delivered in the future. Messaging events can be used to implement synchronization between multiple filaments or to transfer work.

From now on we refer to Timer and Messaging events as *filament events* and differentiate them when necessary.

Workers are responsible for processing events at their scheduled time. Each worker is implemented as a separate thread so that Buttress can take advantage of multiple CPUs. Workers wait until an event is ready to be processed, and based on the event they either issue the I/O in the event, or call the appropriate filament.

The last worker to finish processing an event maintains the time until the next event is ready to be processed. In addition, because we found keeping time using `gettimeofday()` and `usleep()` to be slow and inaccurate, the worker keeps time by spinning; that is, executing a tight loop and keeping track of the time using the CPU cycle counter.

Let us now describe, with a simple example, the functions that a worker performs. We will then translate these worker functions into a generic state transition diagram. We simplify the exposition below for convenience, and in the following section, we discuss specific details needed to achieve higher timing accuracy and throughput.

A worker (*A*) starts by checking if there are events to be processed. Say it found a timer event, and that it was time to process it. If this worker was spinning, then it wakes up a worker thread (*B*) to keep time. Worker *A* then processes the timer event by calling the appropriate filament. Say that the filament generates an I/O to execute in the future. Worker *A* queues it for later processing, and then checks if any events are ready. Since none are ready and worker *B* is spinning, it goes to sleep. Meanwhile worker *B* spins until it is time to process the I/O event, wakes up worker *A* (as before), while *B* issues the I/O. Once the I/O completes worker *B* calls the filament with the completed I/O and goes back to checking for ready events. This procedure continues until there are no events left and there, are no outstanding I/Os to be issued.

We now generalize the above example with generic state transitions (see Figure 3).

1. Check event queue: This is the central dispatch state. In this state, the worker determines if a filament is runnable, or if an I/O is issuable, and transitions to the appropriate state to process the filament or I/O. It also wakes up another worker to replace itself to guarantee someone will be spinning. If no event is ready, the worker either transitions to the spin state or the sleep state based on whether another worker is already spinning.

2. Call Filament: The worker calls the filament when either a timer/messaging event is ready, or when an I/O completes. The filament may generate more events. Once all the ready events are processed, the worker transitions to "Queue Events" state to queue the events the filament generated. The worker may queue events while processing filament events ("early queue") to avoid waiting for all events to get processed for slow filaments.

3. Queue events: In this state, the worker queues events which were generated by a filament. If none of those events are ready, the worker transitions into the "check event queue" state. If any of the events is ready, the worker transitions directly to processing it: either issuing the I/O or calling an appropriate filament.

4. Execute I/O: In this state, the worker executes a ready I/O event. Because implementations of asynchronous I/O on existing operating systems are poor, Buttress uses synchronous I/O, and hence the worker blocks for I/O completion. Once the I/O completes, the worker transitions directly to calling the appropriate filament with the completed I/O event.

5. Spin: A worker starts "spinning" when, after checking the event queue for events to process, it finds that there are no ready events and no other spinning worker.

To prevent deadlock, it is necessary to ensure that not all workers go to sleep. Recall that in Buttress, there is no single thread that is responsible for dispatching events; the functionality is distributed among the workers. Hence if all the workers went to "sleep", there will be a deadlock. Instead, one of the workers always spins, periodically checking if the event at the head of the central queue is ready to be processed.

When the event queue is empty and all other workers are asleep, the spinning worker wakes one thread up and exits; the rest of the workers repeat this process until all threads exit.

4.2 Filament programming interface

There are two ways one can use Buttress: a) configure and run pre-defined library filaments, and b) implement new workloads by implementing new filaments.

Currently Buttress includes filaments that: a) implement different distributions for inter I/O time and device location accessed, b) replay an I/O trace, and c) approximate benchmarks such as TPC-B [21] and SPC-1 [9].

To support programming new filaments, Buttress exports a simple single threaded event-based programming interface. All the complexity of actually scheduling, issuing, and managing events is completely hidden from the filaments. The programmer needs to implement only the logic required to decide what event to generate next. Programmers may synchronize between filaments using message events.

4.3 Statistics gathering

To allow for shared and slow statistics, Buttress uses the same event processing core to pass I/O completion events to filaments which are dedicated to keeping statistics. The set of statistics to keep is specified at run time in a configuration file, which causes Buttress to build up multiple statistic filaments that may be shared by I/O generating filaments.

Some statistics, such as mean and standard deviation are easy to compute, other statistics such as approximate quantiles [16], or recording a full I/O trace can potentially take much longer due to occasional sorting or disk write. For this reason, we separate the steps of generating I/Os, which needs to run sufficiently fast that I/Os always reach the core before their issue time, and statistics, which can be computed independent of the I/O processing. In Buttress, information regarding each I/O is copied into a collection buffer in a filament, without computing the required statistics. Once the collection buffer is full, it is sent to a "statistics thread" using a messaging event. This allows the I/O generation filament to run quickly, and it improves the efficiency of computing statistics because multiple operations are batched together.

5 Key optimizations

The architecture presented in the previous section requires optimization to achieve the desired high throughput and low issue-error. Some of the important implementation questions that need to be addressed are:

- How to minimize latency for accessing shared data structures?
- How to ensure that time critical events get processing priority?
- How to minimize the impact of a non real-time OS with unpredictable system call latencies and preemption due to interrupts?
- How to synchronize timing between the multiple CPUs on an SMP which is required to achieve high throughput?
- How to work around the performance bottlenecks due to the compiler and programming language without sacrificing portability?
- How to identify performance bottlenecks?

In this section, we present some of the techniques we use to address the above questions, and also describe our technique for identifying where optimization is necessary.

5.1 Minimizing latency when accessing shared structures

Shared data structures must be protected by locks. However locks cause trains of workers, contending on the lock, which builds up increasing latency. Additionally, interrupts can force locks to be held longer than expected. Worse, we observed that on Linux, with the default 2.4 threads package, it takes about 10 times longer to release a lock if another thread is waiting on it. Therefore it is important to a) minimize waiting on shared locks, b) minimize the time spent in the critical section, and c) minimize the total number of lock operations. We address the locking problems using *bypass locking* to allow a thread to bypass locked data structures to find something useful to do, reduce the critical section time by pairing priority queues with dequeues, and minimize lock operations using filament event batching and carried events.

Minimizing lock operations

The queues, where workers queue and pick events to process, are shared data structues and accesses to these queues is protected by locks. Hence to reduce the number of lock operations we try to avoid queuing events on these central structures if possible, and attempt to process events in batches.

Workers get new events in the queue-events state or the execute-I/O state, and process events that are ready to be processed in the execute-I/O or call-filament states. To minimize lock operations we enable workers to *carry*, without any central queuing, events that are ready to be processed directly from the queue-events state to the execute-I/O or call-filament states, or execute-I/O to the call-filament state. This simple optimization directly reduces the number of lock operations. Buttress workers prefer to carry I/O events over other events that could be ready, because I/O events are the most time critical.

When processing filament events, workers remove all of the ready events in a single batch; this allows a worker to process multiple filament events with just one lock acquisition (recall that a filament is single threaded and thus locked by the worker executing it). To enable such batch processing, Buttress keeps a separate event queue for each filament rather than placing the events in a central priority queue, which would tend to intermingle events from different filaments. To enable such distributed (per filament) queues, while still allowing for a centrally ordered queue, what is stored centrally is a hint that a filament may have a runnable event at a specified time, rather than the actual event. Workers thus skip hints which correspond to events that have already been processed when working through the central queues.

The same optimization cannot be performed for I/O events because unlike filament events, I/O events cannot be batched – Buttress uses synchronous I/O because we found support for asynchronous I/O inadequate and lacking in performance on stock operating systems. However because I/Os happen frequently and are time critical, we use different queues for the pending hints and pending I/O events, and directly store the I/O events in their own priority queue.

Minimizing critical section time

Though removing an element from a priority queue is theoretically only logarithmic in the length of the queue, when shared between many separate threads in a SMP, each of those operations becomes a cache miss. To alleviate this problem, we pair together a priority queue with a deque, and have a thread move all of the ready events into the deque. This benefits from the fact that, once the queue is searched for a ready event, all the requisite cache lines are already retrieved, and moving another event will cause very few additional cache misses. Removing an entry from the double ended queue only takes at most 2 cache misses: one to get the entry and one to update the head pointer. This combination minimizes the time in critical sections when bursts of events need to be removed from the priority queues.

Bypass locking

While we have minimized the number of lock operations and the time spent in critical sections, at high load it is likely that a thread will get interrupted while holding one of the filament hint or I/O locks. If there are multiple runnable events, we would prefer that the thread remove one of the other events and continue processing, rather than waiting on a single lock, and incurring the high wakeup penalty.

Therefore, we partition the hint queue and the I/O queues. When queuing events, the worker will try each of the queues in series, trying to find one which is unlocked, and then putting events on that one. If all the queues are locked, it will wait on one of the locks rather than spin trying multiple ones. When removing entries, the worker will first check a queue-hint to determine if it is likely that an entry is ready, and if so, will attempt to lock and remove an entry. If the lock attempt fails, it will continue on to the next entry. If it finds no event, and couldn't check one of the possibilities, it will wait on the unchecked locks the next time around.

This technique generally minimizes the amount of contention on the locks. Our measurements indicate that going from one to two or three queues will reduce the amount of contention by about a factor of 1000, greatly reducing the latency of accessing shared data structures. However, at very high loads, we still found that workers were forming trains, because they were accessing the different queues in the same order, so we changed each worker to pick a random permutation order to access the queues; this increases the chance that with three or more queues two workers which simultaneously find one queue busy will choose separate queues for trying next.

We use a similar technique for managing the pool of pages for data for I/Os, except that in this case all threads check the pools in the same order, waiting on the last pool if necessary. This is because we cache I/O buffers in workers, and so inherently have less contention, and by making later page pools get used less, we pre-allocate less memory for those pools.

5.2 Working around OS delays

Buttress is designed to run on stock operating systems and multiprocessor systems, which implies that it needs to work around delays in system calls, occasional slow execution of code paths due to cache misses, and problems with getting accurate, consistent time on multiprocessor systems.

There are three sources of delay between when an event is to be processed and when the event is actually processed: a) a delay in the signal system call, b) a scheduling delay between when the signal is issued and the signaled thread gets to run, and c) a delay as the woken thread works through the code-path to execute the event. Pre-spinning and low priority spinners are techniques to address these problems.

Pre-spin

Pre-spin is a technique whereby we start processing events "early", and perform a short, unlocked spin right before processing an event to get the timing entirely right. This pre-spin is necessary because the thread wake-up,

and code path can take a few 10s of μs under heavy load. By setting the pre-spin to cover $95-99\%$ of that time, we can issue events much more accurately, yet only spin for a few μs. Naturally setting the pre-spin too high results in many threads spinning simultaneously, leading to bad issue error, and low throughput.

Pre-spin mostly covers problems (a) and (c), but we find that unless we run threads as non-preemptable, that even the tight loop of `while(cur_time() < target_time) {}` will very occasionally skip forward by substantially more than the $< 1\mu$s that it takes to calculate cur_time(). This may happen if a timer or an I/O completion interrupt occurs. Since these are effectively unavoidable, and they happen infrequently (less than 0.01% at reasonable loads), we simply ignore them.

Low priority spinners

If the spinning thread is running at the same priority as a thread actively processing events, then there may be a delay in scheduling a thread with real work to do unless the spinning thread calls sched_yield(). Unfortunately, we found that calling sched_yield() can still impact the scheduling delay because the spinning thread is continually contending for the kernel locks governing process scheduling. We found this problem while measuring the I/O latency of cache hits with a single outstanding I/O.

Low priority spinners solve this problem by re-prioritizing a thread as lowest priority, and only allowing it to enter the spin state. This thread handles waking up other threads, and is quickly preempted when an I/O completes because it is low priority and so doesn't need to yield.

Handling multiprocessor clock skew

Typically, in event processing systems, there is an assumption that the different event processing threads are clock synchronized. Though this is always true on a uniprocessor system, clock skew on multiprocessors may affect the system substantially. This is especially tricky when one needs to rely on CPU clock counters to get the current time quickly.

In Buttress, each worker maintains its own time, re-synchronizing its version of the time with gettimeofday() infrequently, or when changes in the cycle counter indicate the worker must have changed CPUs. However, small glitches in timing could result in incorrect execution. Consider the following situation: worker 1 with a clock of 11μs is processing a filament event, when worker 2 with a clock of 15μs tries to handle an event at 15μs. Since the filament is already running, worker 2 cannot process the event, but it assumes that worker 1 will process the event. However worker 1 thinks the event is in the future, and so with the hint removed, the event may never get processed. This tiny 4μs clock skew can result in incorrect behavior. The solution is for workers to mark filaments with their current clock, so that inter-worker clock skew can be fixed. The problem occurs rarely (a few times in a 10+ minute run), but it is important to handle it for correctness.

5.3 Working around C++ issues

One of the well known problems with the standard template library (STL) is the abstraction penalty [20], the ratio of the performance of an abstract data structure to a raw implementation. We encountered the abstraction penalty in two places: priority queues and double-ended queues. The double ended queue is implemented with a tree, which keeps the maximum operation time down at the expense of slower common operations. Using a standard circular array implementation made operations faster at the expense of a potentially slow copy when the array has to be resized. Similarly, a re-implementation of the heap performed approximately $5\times$ faster than STL for insertion and removal when the heap is empty, and about $1.2\times$ faster when the heap is full (on both HP-UX and Linux with two different compilers each). The only clear difference between the two implementations was that STL used abstraction much more (inserting a single item nested about eight function calls deep in the STL, and one in the rewrite).

Other performance problems were due to operations on `long long` type, such as *mod* and conversion to `double`. The mod operation was used in quantization; our solution was to observe that the quantized values tend to be close to each other, and therefore, we calculate a delta with the previous quantized value (usually only 32-bits long) and use the delta instead followed by addition.

5.4 Locating performance bottlenecks

Locating bottlenecks in Buttress is challenging because many of them only show up at high loads. We addressed this with two approaches. First, we added counters and simple two-part statistics along many important paths. The two part statistics track "high" and "low" values separately for a single statistic, which is still fast, and allows us to identify instances when variables are beyond a threshold. This is used for example to identify the situations when a worker picks up an event from the event queue before the event should happen or after; or the times when few (say less than 75%) of the workers are active.

Second, we added a vector of (time, key, value) trace entries that are printed at completion. These trace entries allow us to reconstruct, using a few simple scripts, the exact pattern of actions taken at runtime. The vectors are

per worker, and hence lock-free, leading to low overhead when in used. The keys are string pointers, allowing us to quickly determine at runtime if two trace entries are for the same trace point, and optionally collapse the entries together (important, for example, for tracing in the time-critical spin state).

The counters and statistics identify which action paths should be instrumented when a performance problem occurs, and the trace information allows us to identify which parts of those paths can be optimized.

6 Experimental evaluation

In this section, we present experimental results concerning I/O issue speed, I/O issue error, and overhead of Buttress for a wide variety of workloads and storage subsystems. We also compare characteristics of the generated workload and that of the original to determine the fidelity of the trace replay.

6.1 Experimental setup

We used three SMP servers and five disk arrays covering a wide variety of hardware. Two of the SMP servers were HP 9000-N4000 machines: one with eight 440MHz PA-RISC 8500 processors and 16GB of main memory (**timpani**), the other with two 440MHz PA-RISC 8500 processors and 1 GB of main memory (**bongo**). The third was an HP rp8400 server with two 750MHz PA-RISC 8500 processors and 2 GB of main memory (**piccolo**). All three were running HP-UX 11.0 as the operating system.

We used five disk arrays as our I/O subsystem: two HP FC-60 disk arrays [11], and one HP XP512 disk array [2], one HP XP1024 disk array [1], and one HP VA7400 disk array [3]. Both the XP512 and XP1024 had in-use production data on them during our experiments.

The XP1024 is a high end disk array. We connected timpani directly to front-end controllers on the XP1024 via eight 1 GBps fibre-channel links, and used two back end controllers each with 24 four-disk RAID-5 groups. The array exported a total of 340 14 GB SCSI logical units spread across the array groups for a total of about 5 TB of usable disk space.

Piccolo was connected via three 1 GBps links to a Brocade Silkworm 2400 fibre-channel switch, that was also connected to the XP512 and VA7400. The XP512 used two array groups on one back-end controller exporting 17 logical units totaling 240 GB of space. The VA7400 is a mid-range virtualized disk array that uses AutoRaid [24] to change the RAID level dynamically, alternating between RAID-1 and RAID-6. It exported 50 virtual LUs, each 14 GB in size for a total of 700 GB of space spread across 48 disks.

Bongo was connected to two mid-range FC-60 disk arrays via three 1GBps fibre channel links to a Brocade Silkworm 2800 switch. The FC-60 is a mid-range disk array; one exported 15 18GB 2-disk RAID-1 LUs, and the other 28 36GB 2-disk RAID-1 LUs for a total of 1300 GB of disk space.

6.2 Workloads

We used both synthetic workloads and two application traces: a file server containing home directories of a research group (*harp*), and an e-mail server for a large company (*omail*). In order to create controlled workloads for our trace replay experiments, we also used a modified version of the PostMark benchmark (*postmark*).

The synthetic workload consisted of uniformly spaced 1KB I/Os, issued to 10 logical units spread over all of the available paths; the workload is designed so that most of the I/Os are cache hits. We use *timpani* as the host and the XP1024 disk array as the storage system for the experiments that use this workload.

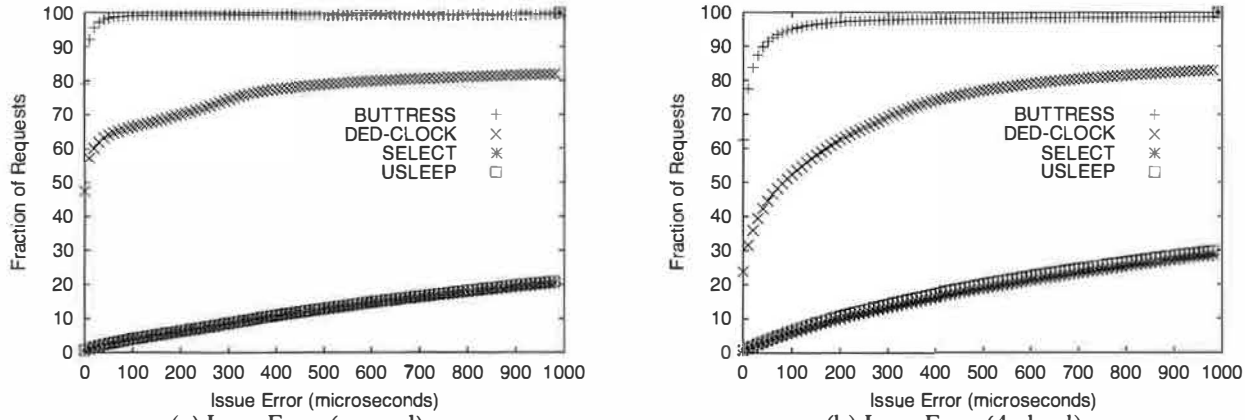
The file-system trace (*harp*) represents 20 minutes of user activity on September 27, 2002 on a departmental file server at HP Labs. The server stored a total of 59 file-systems containing user home directories, news server pools, customer workload traces, HP-UX OS development infrastructure, among others for a total of 4.5 TB user data. This is a typical I/O workload for a research group, mainly involving software development, trace analysis, simulation, and e-mail.

The *omail* workload is taken from the trace of accesses done by an OpenMail e-mail server [10] on a 640GB message store; the server was configured with 4487 users, of whom 1391 were active. The *omail* trace has 1.1 million I/O requests, with an average size of 7KB.

The PostMark benchmark simulates an email system and consists of a series of transactions, each of which performs a file deletion or creation, together with a read or write. Operations and files are randomly chosen. We used a scaled version of the PostMark benchmark that uses 30 sets of 10,000 files, ranging in size from 512 bytes to 200KB. To scale the I/O load intensity, we ran multiple identical copies of the benchmark on the same file-system.

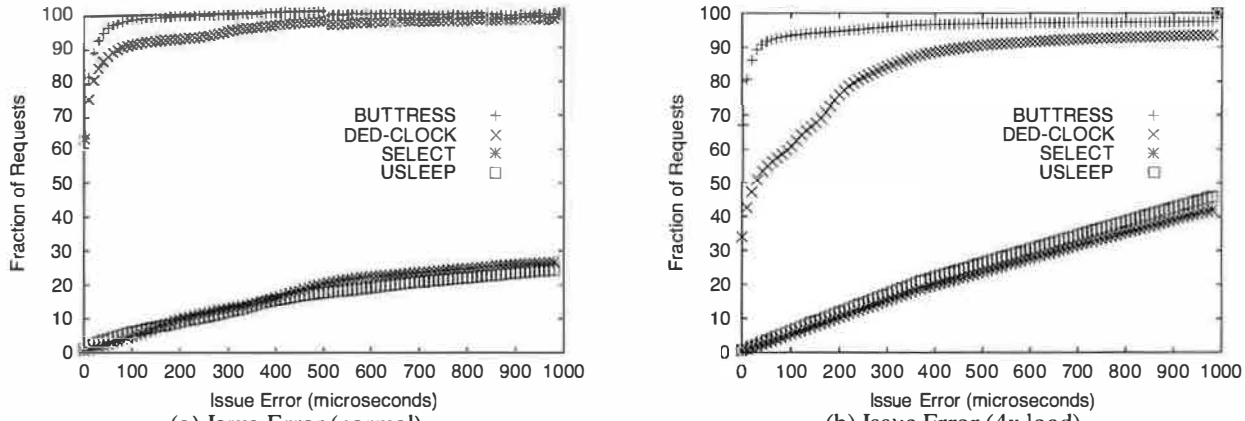
6.3 I/O issue error

We now present a detailed analysis of various trace replay schemes, including Buttress, on their behavior to achieve good timing accuracy as the I/O load on the system changes for a variety of synthetic and real application workloads. In Section 2, we demonstrated that the issue error impacts the workload characteristics; in this section, we focus on the issue error itself.



(a) Issue Error (normal) (b) Issue Error (4x load)

Figure 4: Issue error for the omail trace when replayed on timpani with the XP1024.



(a) Issue Error (normal) (b) Issue Error (4x load)

Figure 5: Issue error for the harp trace when replayed on timpani with the XP1024.

In Figures 4 and 5 we plot the CDF of the issue error for Buttress, DED-CLOCK, SLEEP, and SELECT using the harp and the omail workload. We use two variants of these workloads: we replay the workload at the original speed and quadruple the speed. This lets us quantify the issue error as the throughput changes. These experiments were performed on *timpani* using the XP1024 disk array.

These results show that Buttress issues about 98% of the I/Os in the omail workload within 10 μs of the actual time in the original workload and 95% of the I/Os in the harp workload within less than 50 μs of their actual time. On the other hand, OS-based trace replay mechanisms fare worst: both SLEEP and SELECT could achieve 1 millisecond of issue accuracy for only about 30% of the I/Os in either workload. The DED-CLOCK was slightly better, issuing 89% of the I/Os in the harp trace and 60% of the I/Os in the omail trace within 100 μs of their intended time. This is because DED-CLOCK can more accurately keep time using the CPU cycle counters, but overwhelmed by the thread wakeup overhead when dealing with moderate I/O rates.

The results with the faster replays indicate that Buttress continues to achieve high I/O issue accuracy for moderate loads: 92% of the I/Os in the harp workload and the 90% of the I/Os in the omail workload are issued within 50 μs of their intended issue times. An interesting observation is that the SLEEP and SELECT based mechanisms perform slightly better at higher load (4x issue rate) than at lower loads. This is because in the higher load case, the kernel gets more opportunities to schedule threads, and hence more I/O issuing threads get scheduled at the right time. The dedicated clock-thread based approach, however, is restrained by the speed at which the clock-thread can wake up worker threads for I/O issue – especially for the omail workload where the I/O load steadily runs at a moderate rate.

Figure 6 shows the issue error for the harp workload when we use the 2-processor server bongo and the two mid-range FC-60 disk arrays. While this environment has sufficient resources to handle the steady state workload,

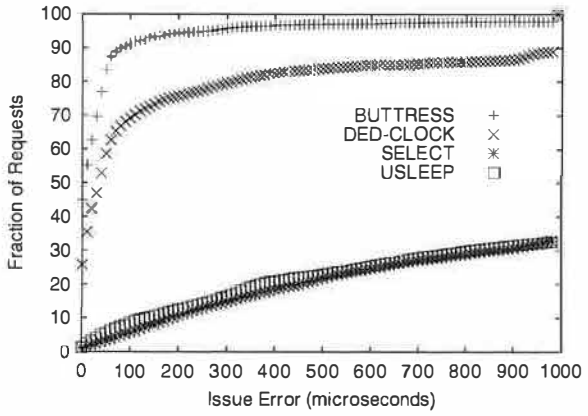


Figure 6: Issue error for harp trace on two-processor bongo server, using two mid-range FC-60 disk arrays.

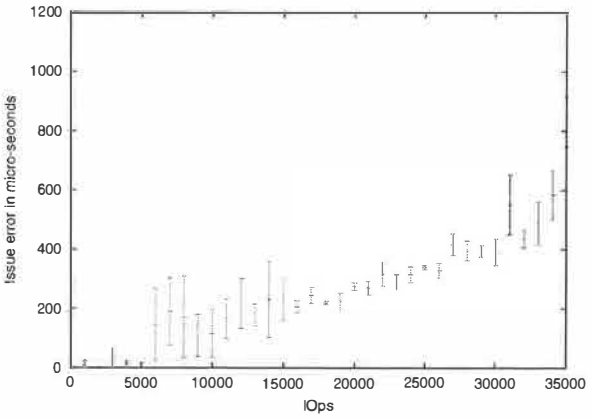


Figure 7: Issue error of Buttress as a function of the number of I/Os per second issued.

it does not have enough resources to handle the peaks. When the arrays fall behind in servicing I/Os, contention occurs; as a result, both Buttress and DED-CLOCK show heavy-tailed issue error graphs. Also, having only two processors to handle all system events introduces additional delays when interrupts are being processed.

We also used synthetic, more regular workloads to determine how accurately Buttress issues I/Os as the load increases. We measured the difference between the time that the I/O was supposed to be issued and the time when it was actually issued. The results presented are averages of 3 runs of the experiments using 3 different sets of 10 destination devices. Figure 7 presents the results, with the issue error being plotted against the number of IOPS performed by Buttress. We use IOPS because it correlates with the number of events that Buttress needs to handle.

Another measure of Buttress' performane in terms of its overhead, is whether Buttress can get throughputs comparable to those of I/O generation tools specifically engineered to generate only a particular pattern of I/Os. To answer this question we wrote a special-purpose program that uses multiple threads issuing I/Os using pread() to each of the available devices. We used timpani with the XP1024 for these experiments, and noticed that the maximum throughput we could achieve using the special-purpose program was 44000 IOPS (issuing I/Os to cached 1KB blocks). On the same hardware and setup, Buttress could issue I/Os at 40000 IOPS, only 10% less.

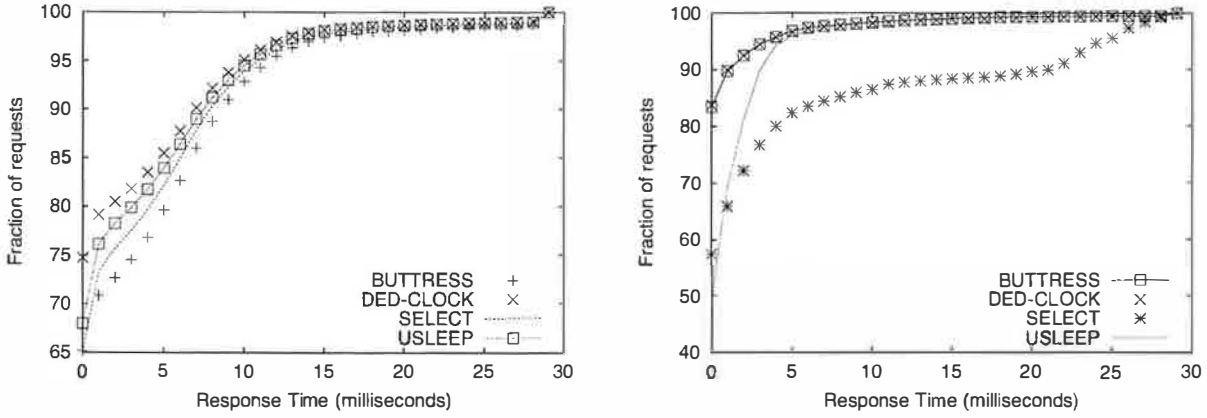
6.4 Workload fidelity

In this section, we examine the characteristics of the I/O workloads produced using trace replay and expand our discussion in Section 2. We focus on two characteristics of the I/O workload: response time and burstiness – Figure 8 (the detailed version of Figure 2(a)) presents the CDF of the measured response times across various trace replay mechanisms; and Figure 10 compares the burstiness characteristics of the original workload with the burstiness characteristics of the workload generated by Buttress. Figure 10 visually shows that Buttress can mimic the burstiness characteristics of the original workload, indicating that Buttress may be "accurate enough" to replay traces.

For the omail workload, all of the trace replay mechanisms are comparable in terms of the response time of the produced workload: the mean response times were within 15% of each other. For this trace, even though the I/Os were issued at a wide range of accuracy, the effects on the response time characteristics were not substantial. This is not so for the harp workload – different trace replay mechanisms produce quite different response time behavior. This is partially explained by the high-burstiness exhibited in the harp workload; sharp changes in the I/O rate are difficult to reproduce accurately.

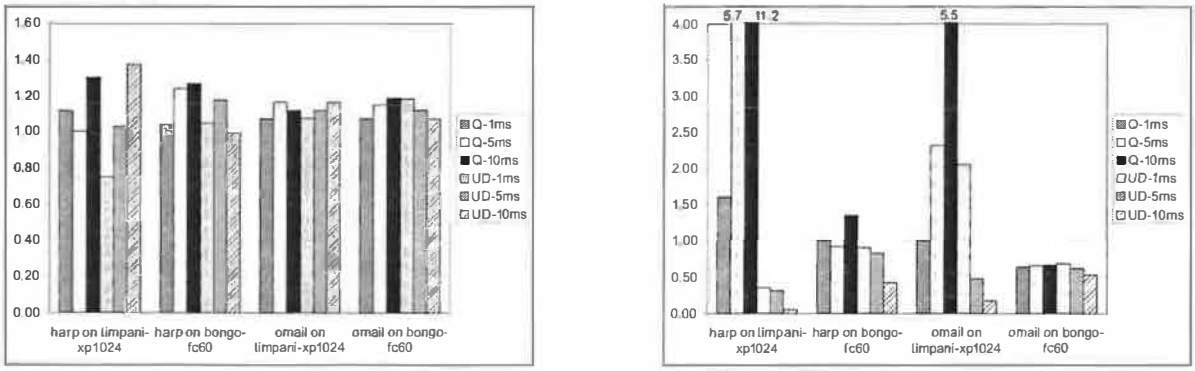
In order to understand the impact of I/O issue accuracy on the application I/O behaviour, we studied the effect of controlled issue error using two means: a) by introducing a uniform delay to the issue times of each I/O and b) by quantizing the I/O issue times around simulated scheduling boundaries. Figure 9 shows the results of the sensitivity experiments for two application metrics, response time and burstiness. It shows that the mean response time changes as much as 37% for the harp workload and 19% for the omail workload. The effects of issue error on the burstiness characteristics (mean queue size) is more dramatic: as much as 11 times for the harp workload and five times for the omail workload. This shows that the bursty workloads are more sensitive to the delays in I/O issue times leading to modify their I/O behavior.

So far, we used application workloads collected on different systems; we now look at the PostMark workload and present its characteristics from the trace replays



(a) omail trace (b) harp trace

Figure 8: Response time CDF of various trace-replay mechanisms for harp and omail traces on timpani with XP1024.



(a) Response time (b) Queue size

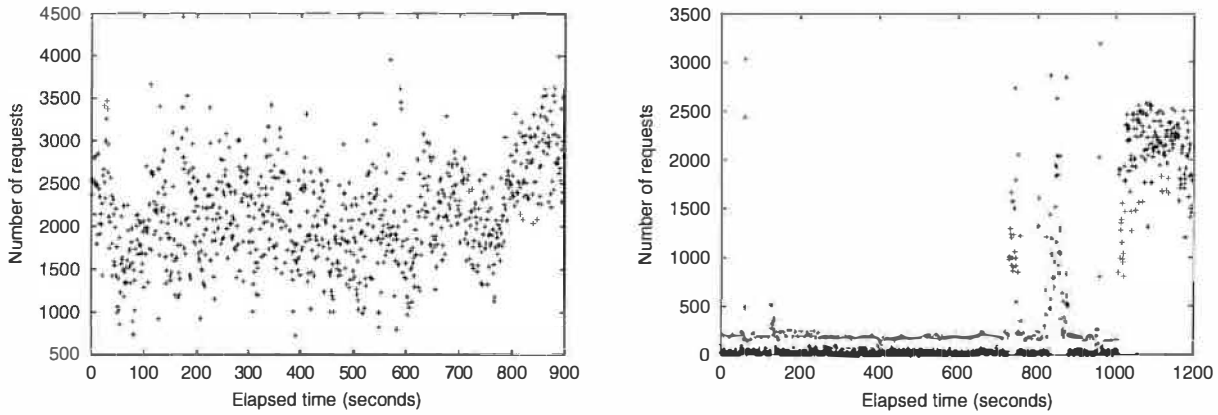
Figure 9: Sensitivity analysis for the impact of I/O issue accuracy (normalized to Buttress) on the application I/O behavior on various systems. All the numbers are normalized to the value of the metric reported by Buttress. Q-X denotes the quantization at X ms boundaries and UD-X denotes the random delay added using uniform distribution with mean X ms.

when we use the same host and the array to replay trace as we used running PostMark. Figure 11 shows the response time characteristics of the PostMark workload on the XP1024 measured from the workload and from the trace replays. The high-accuracy of Buttress helps it to produce almost the exact response time statistics as the actual workload, while the less accurate mechanisms deviate significantly more.

7 Related Work

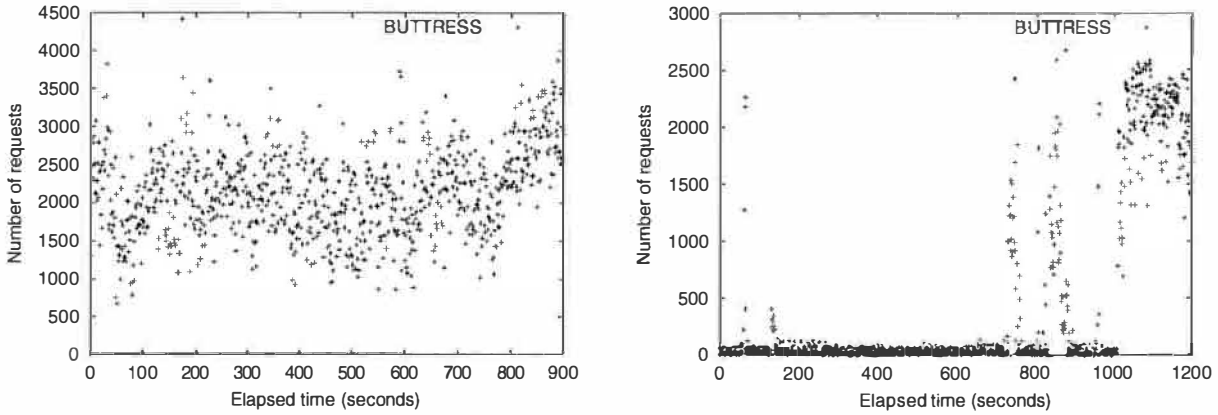
Several benchmarks attempt to emulate the real application behavior: TPC benchmarks [22] emulate common database workloads (e.g., OLTP, data warehousing), Postmark [15], SPECsfs97 [8], and Andrew [12] emulate file system workloads. The I/O load generated from these benchmarks still uses the real systems, e.g., a relational database or a UNIX file system, but the workload (e.g., query suite, file system operations) are controlled in the benchmark. In practice, setting up infrastructures for some of these benchmarks is complex and frequently very expensive; Buttress complements these benchmarks as a flexible and easier to run I/O load generation tool, which does not require expensive infrastructure.

A variety of I/O load generators measure the I/O systems behavior at maximum load: Bonnie [6], IOBENCH [25], IOmeter [13], IOstone [19], IOzone [14], and lmbench [17]. While Buttress could also be used to determine the maximum throughput of a system, it has the capability to generate complex workloads with think-times and dependencies (e.g., SPC-1 benchmark [9] and



(a) omail trace (original)

(b) harp trace at 4X it's original rate (original)



(a) omail (replay)

(b) harp trace at 4X it's original rate (replay)

Figure 10: Burstiness characteristics The X axis is the number of seconds past the start of the trace, and the Y axis is the number of requests seen in the previous 1 second interval. These experiments were run on timpani with the XP1024

TPC-B [21]) and can be used in trace replays. In addition, many of these benchmarks can easily be implemented on top of the Buttress infrastructure, due to its portability, flexibility, and high-performance. Moreover, Buttress can handle general open and closed I/O workloads in one tool.

Fstress [5], a synthetic, flexible, self-scaling [7] NFS benchmark has a load generator similar to the one in Buttress. While the Fstress load generator specifically targets NFS, Buttress is general purpose and can be tailored to generate a variety of I/O workloads. Furthermore, we expect that extending Fstress's "metronome event loop" in a multi-processor environment will face the same set of design issues we address in this paper.

Several papers [18, 23, 4] have been written on programming models based on events and threads, and they make a case for one or the other. The architecture of Buttress can be viewed as using both models. In particular, Buttress uses event-driven model implemented with threads. Buttress uses pthreads so that it can run on SMPs, and multiplexes event-based filaments across them to support potentially millions of filaments each temporarily sharing a larger stack space. Since Buttress is implemented in C++, and C++ facilitates state packaging, we have not found that it poses an issue for us as other researchers have found for implementations in C.

8 Conclusions

We presented Buttress, an I/O generation tool that can be used to issue pre-recorded traces accurately, or generate a synthetic I/O workload. It can issue almost all I/Os within a few tens of ∝s of the target issuing time, and it is built completely in user space to improve portability. It provides a simple interface for programmatically describing I/O patterns which allows generation of complex I/O patterns and think times. It can also replay traces accurately to reproduce workloads from realistic environments.

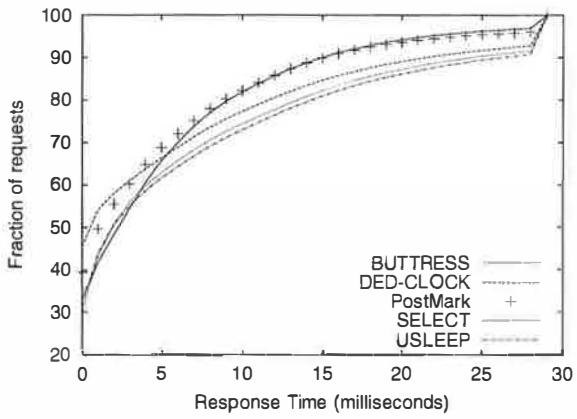


Figure 11: Response time characteristics of the Postmark benchmark and replaying its trace when run on timpani with the XP1024.

9 Acknowledgements

We thank our shepherd Fay Chang for her help in making the presentation better, and the anonymous referees for their valuable comments. We also thank Hernan Laffitte forthe substantial help in setting up the machines and arrays so that the experiments could actually be run.

References

[1] HP StorageWorks disk array xp1024. http://www.hp.com/products1/storage/products/disk_arrays/highend/xp1024/.

[2] HP StorageWorks disk array xp512. http://www.hp.com/products1/storage/products/disk_arrays/highend/xp512/.

[3] HP StorageWorks virtual array 7400. http://www.hp.com/products1/storage/products/disk_arrays/midrange/va7400/.

[4] A. Adya, J. Howell, M. Theimer, W.J. Bolosky, and J.R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proceedings of the USENIX 2002 Annual Technical Conference*, June 2002.

[5] D. Anderson and J. Chase. Fstress: a flexible network file system benchmark. Technical Report CS-2002-01, Duke University, January 2002.

[6] T. Bray. Bonnie benchmark. http://www.textuality.com/bonnie, 1988.

[7] P. Chen and D. Patterson. A new approach to I/O performance evaluation – self-scaling I/O benchmarks, predicted I/O performance. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 1–12, May 1993.

[8] Standard Performance Evaluation Corporation. SPECSFS release 3.0 run and report rules, 2001.

[9] Storage Performance Council. SPC-1 benchmark. http://www.storageperformance.org, 2002.

[10] Hewlett-Packard. HP OpenMail. http://www.openmail.com/cyc/om/50/index.html.

[11] Hewlett-Packard Company. *HP SureStore E Disk Array FC60 - Advanced User's Guide*, December 2000.

[12] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and performance in a distributed file system. *ACM Trans. on Computer Systems*, 6(1):51–81, February 1988.

[13] IOmeter performance analysis tool. http://developer.intel.com/design/servers/devtools/iometer/.

[14] IOzone file system benchmark. www.iozone.org, 1998.

[15] J. Katcher. Postmark: a new file system benchmark. Technical Report TR-3022, Network Appliance, Oct 1997.

[16] G.S. Manku, S. Rajagopalan, and B.G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of data*, pages 426–435, 1998.

[17] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proc. Winter 1996 USENIX Technical Conference*, pages 279–84, January 1996.

[18] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference, January 1996. http://home.pacbell.net/ouster/threads.ppt.

[19] A. Park and J.C. Becker. IOStone: a synthetic file system benchmark. *Computer Architecture News*, 18(2):45–52, June 1990.

[20] A.D. Robison. The Abstraction Penalty for Small Objects in C++. In *Parallel Object-Oriented Methods and Applications '96*, Santa Fe, New Mexico, February 1996.

[21] The transaction processing performance council. TPC Benchmark B. http://www.tpc.org/tpcb/spec/tpcb_current.pdf, June 1994.

[22] Tpc – transaction processing performance council. www.tpc.org, Nov 2002.

[23] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proc. of the 9th Wkshp. on Hot Topics in Operating Systems (HotOS IX)*, pages 19–24, 2003.

[24] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The hp autoraid hierarchical storage system. In *Proc 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 96–108, 1995.

[25] B.L. Wolman and T.M. Olson. IOBENCH: a system independent IO benchmark. *Computer Architecture News*, 17(5):55–70, September 1989.

Designing for disasters

Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, John Wilkes
Hewlett-Packard Laboratories, Palo Alto, CA and Duke University, Durham, NC
{kimberly.keeton, cipriano.santos, dirk.beyer, john.wilkes}@hp.com, chase@cs.duke.edu

Losing information when a storage device or data center fails can bring a company to its knees—or put it out of business altogether. Such catastrophic outcomes can readily be prevented with today's storage technology, albeit with some difficulty: the design space of solutions is surprisingly large, the configuration choices are myriad, and the alternatives interact in complicated ways. Thus, solutions are often over- or under-engineered, and administrators may not understand the degree of dependability they provide.

Our solution is a tool that automates the design of disaster-tolerant solutions. Driven by financial objectives and detailed models of the behaviors and costs of the most common solutions (tape backup, remote mirroring, site failover, and site reconstruction), it appropriately selects designs that meet its objectives under specified disaster scenarios. As a result, designing for disasters no longer needs to be a hit-or-miss affair.

1 Motivation

> *"The cascading blackout that swept through cities from Detroit and Cleveland to Ottawa and New York City was a harsh reminder to enterprises—to companies of all sizes, in fact—that disaster-prevention and business-continuity plans should be at the top of the priority list."* – Techweb, August 22, 2003

Hardware breaks. Software has defects. Viruses propagate. Buildings catch fire. Power fails. People make mistakes. Although we prefer that these events never occur, it is only prudent to defend against them. The cost of data unavailability can be large: a quarter of the respondents to a 2001 survey [EagleRock2001] estimated their outage costs as more than $250 000/hour, and 8% estimated them as more than $1M/hour. The price of data loss is even higher. Recent high-profile disasters have raised awareness of the need to plan for recovery or continuity: since data is a critical asset, the key challenge is to construct *dependable* storage systems that protect data and the ability to access it.

Figure 1 illustrates the most commonly-used alternatives to protect and recover stored data. In all cases, a *primary copy* of the data is protected by making one or more *secondary copies*, which are isolated from failures that may affect the primary copy. Recovery involves either site *failover* to a secondary mirror, or data *reconstruction* at the primary site or a secondary site.

Each technique provides some of the necessary protection; combined, they create a large and complex design space. For example, a popular combination combines internally-redundant disk arrays (RAID), remote mirroring, snapshot and backup to tape: RAID protects against disk failures, mirroring guards against site failures, snapshots address user errors, and tape backup protects against software errors and provides archival storage.

These building blocks for data protection are proven technologies, and they continue to be improved. Even so, it is difficult to combine and configure them to create well-engineered data dependability solutions. No solution can ever provide an absolute guarantee of

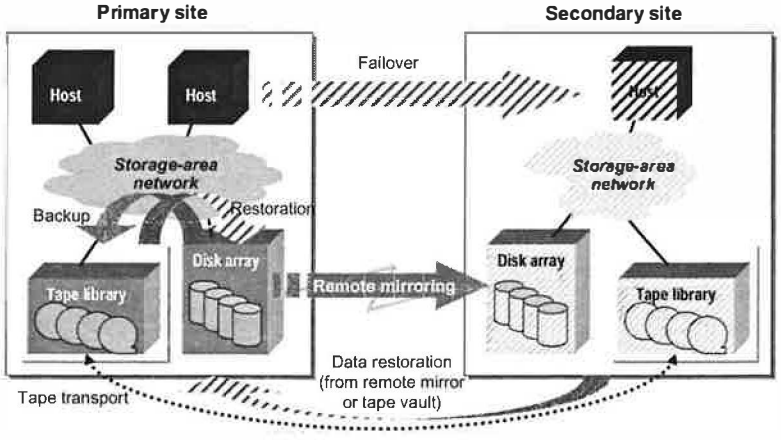


Figure 1: an overview of the disaster recovery alternatives considered in this paper. The standard solutions include inter-array mirroring (local and remote, synchronous and asynchronous), tertiary storage (e.g., tape) backup, remote vaulting, and snapshots, combined with recovery by failover or data reconstruction.

safety, so designers must trade off the desire for rapid recovery and minimum data loss against each solution's price, operational costs, and performance impact across a range of configuration choices. Over-engineered solutions may incur excessive costs to defend against negligible risks. Under-engineered solutions have their own costs in a disaster: crippling outages, loss of critical data, or unacceptably degraded service. Faced with these choices, designers often resort to *ad hoc* attempts to strike the right balance, guided by rules of thumb and their own limited (or even irrelevant) experience.

The key contribution of this paper is to show how to design data dependability solutions *automatically*, given specifications of workload properties, system components, data value, and expected rates of primary copy failures. We describe and evaluate a design tool that combines quantitative models of the standard protection and recovery techniques (depicted in Figure 1) with an off-the-shelf optimizing solver. The output of the tool is a cost-effective data dependability design.

The paper makes three further contributions, each of which is essential to automating dependability design:

1. *Dependability metrics.* We describe metrics to specify data dependability requirements, covering both data *reliability* (i.e., the absence of data loss or corruption) and data *availability* (i.e., the ability to access data when desired). These metrics characterize the severity of a failure in terms of the *financial* cost of the resulting data loss and service outages.
2. *Models of data protection alternatives.* We develop quantitative models of the dependability properties and costs of common disaster tolerance solutions.
3. *Optimization framework.* We show how to formulate data dependability design as an efficient optimization problem to balance cost and risk exposure—precisely the kind of task that automation can aid. The optimization objective is to minimize the sum of *outlays* and the financial *penalties* for failures.

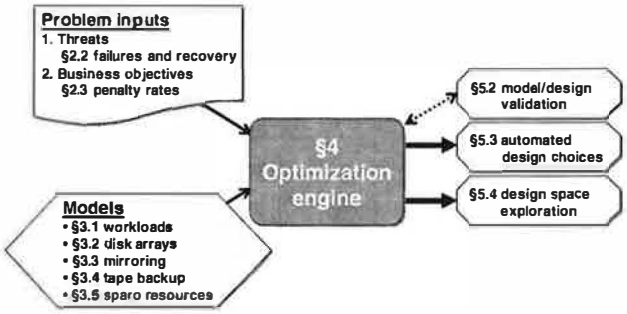


Figure 2: structure of the dependability design tool. Section numbers on the graphic indicate the sections that discuss each topic in the text.

2 Overview

To automate the design process, we must first capture the design objectives in a quantitative way. The formulation must be solution-independent, which means it specifies only the goals, and not the solutions. "Backup my data once a day" describes an implementation, rather than the underlying objective. A better choice is "lose no more than 24 hours' updates." Better still is "minimize my total costs if each hour of lost updates costs me $500,000."

In a business context, most choices come down to money. A key premise of our work is that effective dependability design must meet concrete objectives specified in financial terms. This section explains our financial framework for specifying dependability goals and configuring cost-effective solutions for data protection and recovery.

2.1 Failures and recovery

Data dependability solutions protect against several threat categories:

Data loss: threats that cause data to be lost, including failures of the storage devices themselves, or an encompassing failure, such as a building fire. Recent updates may be more vulnerable to loss because they may not have propagated to fully protected storage (e.g., they might reside in a volatile OS buffer, or might not have propagated to a remote site yet).

Data corruption: threats that change data into a form that cannot be used. Corruption may result from user error, defective software or firmware, and attacks.

Data inaccessibility: threats that do not damage data, but prevent access to it, such as failure of a network link, switch, or disk array controller.

To limit the scope of this study, we focus on data loss events for the primary copy, such as a primary site disaster or failure of a primary disk array. Data corruption and inaccessibility threats can be mapped into loss of the primary copy. We leave a treatment of secondary failures to future work.

After a failure, normal operation resumes when either of two states is achieved:

1. *Failover:* the secondary is usable as the primary.
2. *Reconstruction:* the primary is restored and usable again, either at its original site or at another site outside of the scope of the failure.

Our tool considers only the minimum repairs needed to restore service. In practice, repair is not fully complete until the original level of redundancy is restored, so the secondary can tolerate another failure. In the case of

failover, typically a planned *fail-back* operation restores each site or copy to its pre-failure role.

2.2 Recovery time and recovery points

Business continuity practitioners currently use two metrics to capture data dependability objectives:

1. *Recovery time.* The *Recovery Time Objective (RTO)* specifies the maximum allowable delay until application service is restored after a failure event. The RTO can range from seconds to days.
2. *Data loss.* Recovery may require reverting to some consistent point prior to the failure, discarding updates issued after that recovery point. The *Recovery Point Objective* (RPO) gives the maximum allowable time window for which recent updates may be lost. The RPO can range from zero (no loss is tolerable) to days or weeks.

Discussions with storage designers suggest that the dependability objectives for the repair step are to restore the original level of redundancy "soon." How soon and at what expense are less well-defined: best-effort solutions are generally used.

2.3 Penalty rates

Choosing the right RTO and RPO requires understanding (1) the business costs of service outages and data loss *and* (2) the recovery behavior and cost of all available solutions to achieve each RTO and RPO. This process is a complex optimization problem in a large design space, and is hard to do well by hand.

Instead of fixed RTO and RPO targets, our approach quantifies the financial impacts of outages and data loss as *penalty rates*, expressed as $/hour. The *loss penalty rate* specifies the cost per hour of lost updates. The *outage penalty rate* specifies the cost per hour of service interruption. Our design tool uses these rates to balance expected failure impacts against outlays to arrive at a cost-effective solution. The tool identifies the RTO and RPO as a *side effect* of this design process.

Determining penalty rates is still not easy, but it can be done using techniques such as Business Impact Analysis (BIA), and tools (e.g., http://www.availability.com) that help managers quantify their exposure to outages and data loss. The insurance industry regularly assesses threats and exposures in many other domains of equivalent complexity. Assessing these penalty rates is a strictly simpler and less error-prone task than determining the right RTO and RPO values.

Even though site and regional failures are extremely rare in North America, government regulation or liability obligations [CNT2003] may mandate specific maximum RPO and RTO values. More stringent requirements are particularly likely in the financial industry [SEC2002]. Mandated values can be specified as bounds for the RPO and RTO results obtained from our approach, although the bounds are rarely tight.

2.4 Putting it all together

Our design tool works as follows. First, it uses the models described in Section 3 to explore the solution space—including the ranges of configuration parameter settings for each alternative—and to predict the deployment outlays, worst-case recovery time, and worst-case data loss for each candidate solution. The tool then selects the best design alternative (e.g., synchronous mirroring), together with the best values for its configuration parameters (e.g., two wide area links) and the best recovery alternative (e.g., failover to the secondary site). This optimal configuration balances the system cost (outlays) with the worst-case financial costs (penalties) for expected primary failure events.

3 Modeling protection techniques

This section outlines the models and parameters for the data protection and recovery alternatives considered in this paper. Each model defines a set of input parameters (e.g., component outlays and performance attributes) and their values, a set of configuration parameters (e.g., number of tape drives or network links to a remote mirror) and their ranges of valid settings, and equations relating the parameters to the measures of interest: worst-case data loss and recovery time. It is trivial to model different parameter settings for system components, and it is straightforward in principle to incorporate new data protection techniques into the design tool.

To normalize costs, all capital outlays are depreciated linearly over three years, and all recurring costs (e.g., leasing and support costs) are given at an annual rate.

3.1 Workload characteristics

Data dependability designs are sensitive to characteristics of the storage workload. Many data protection schemes are sensitive to the *average update rate*: the volume of updates over a given interval, divided by the interval length. Techniques that accumulate modifications over an interval (e.g., incremental tape backup) are more sensitive to the workload's *unique update rate*, the number of unique data items written over an interval. Longer accumulation intervals allow more time for overwrites, so they often have lower unique update rates. We model this rate by a series of tuples of the form <interval duration, unique update rate>. Synchronous mirroring solutions are also sensitive to the

short-term peak-to-average burstiness of writes, which is typically 3–10· the long-term average update rate.

These characteristics can be measured from an existing system or estimated by a person or a tool from a repertoire of well-known workloads. Table 1 quantifies the relevant workload characteristics for the cello2002 workload [Ji2003], a publicly available trace of a time-sharing system at HP Labs. The experiments reported in Section 5 use this workload.

Parameter	Description	Values
wkldCapacity	overall workload capacity	1.36 TB
avgUpdateRate	average update rate; no rewrite absorption	799 KiB/s
burstMultiplier	short-term burst update-rate multiplier	10·
uniqueUpdate-Rate(duration)	unique update rate over the given interval, after absorbing overwrites: <interval duration, unique update rate>	<1 min, 727KiB/s> <5 min, 689KiB/s> <1 hr, 581KiB/s> <4 hr, 458KiB/s> <12 hr, 350KiB/s> <24 hr, 317KiB/s> <48 hr, 317KiB/s>
uniqueCapacity (duration)	total size (capacity) of unique updates during given duration	*duration · uniqueUpdate-Rate(duration)*

Table 1: workload attributes for cello2002.

We ignore workload characteristics that do not affect the choice of dependability solutions. Existing tools can address these design issues (e.g., [Anderson2002a]).

3.2 The primary copy

We assume one or more disk arrays store the primary copy of data. We assume that they protect data against internal single-component failures, and consider only complete failure of the primary array or site. For simplicity, we consider the case where the entire dataset is protected in the same way; in practice, different storage volumes may be protected differently.

Our disk array cost model is based on a diverse set of disk arrays from Hewlett-Packard, including the HSG80, EVA and XP1024 arrays. The model captures details such as the costs of the array chassis/enclosures, redundant front-end controllers (including caches), XP1024 back-end controllers, and the disk drives and the trays in which they are mounted. It estimates the cost of floor space, power, cooling, and operations by using a fixed facilities cost plus a variable cost that scales with capacity. For the experiments reported here we used RAID-10 configurations of the EVA model summarized in Table 2.

Parameter	Description	Values
maxDisks	upper bound on number of disks in each array	256
diskCapacity	capacity per disk drive	73 GB
arrayCacheCapacity	array cache capacity	32 GiB
arrayReloadBW	maximum disk array reload (restore) rate	512 MB/s
enclosureCost	outlay cost of disk array enclosure	$189 890 / enclosure
diskCost	outlay cost of disk	$3549 / disk
fixedFacilitiesCost	fixed outlay cost	$60k / year
varFacilitiesCost	variable outlay cost	$1 / GB / year
depreciationPeriod	amortization period for capital outlay costs	3 years

Table 2: primary copy model parameters for the HP EVA disk array and facility costs. These EVA costs were representative at the end of 2003, and may not be actual list prices. The facility costs are estimates.

The overall annualized outlay cost for disk array storage and facilities for the primary copy is:

$$primaryCost = \frac{(numDiskArrays \cdot enclosureCost + numDisks \cdot diskCost)}{depreciationPeriod} + varFacilitiesCost \cdot wkldCapacity + fixedFacilitiesCost$$

Parameter	Description	Values
mirrorCache-Capacity	size of buffer used to smooth update bursts	100 MiB
$interval_{asyncB}$	asyncB batch duration for coalescing writes	1 min, 5 min, 1 hr, 4 hr, 12 hr, 24 hr
linkBW	link bandwidth	T3: 6MiB/s OC3: 16MiB/s
$links_{max}$	upper bound on number of links	16
linkCost	annual outlay cost for link	T3: $60 000 / year OC3: $456 000 / year

Table 3: parameters for the remote mirroring model. Link cost estimates come from [Ji2003].

3.3 Remote mirroring

Remote mirroring protects against loss of the primary by keeping an isolated copy on one or more disk arrays at a secondary site. Our remote mirroring model includes a transfer rate (bytes/s) for the network link connecting the mirrors, a link cost ($/year), and an upper bound on the number of links that may be deployed. We currently model T3, OC3, OC48, and local and metropolitan-distance (10km) optical FibreChannel links. Adding new link types is as simple as specifying

their parameter values. Table 3 shows the parameters for the T3 and OC3 link types used in the experiments.

The primary cost factors for remote mirroring systems are the costs of (1) the storage for the remote copy and (2) the network links required to match the write rate at the primary. The costs and penalties depend on the remote mirroring protocol [Ji2003]:

- *Synchronous mirroring (sync)*: the secondary receives and applies each write before the write completes at the primary. This scheme requires low latency (e.g., close proximity) between the sites to obtain good performance, but no data is lost if the primary fails: $dataLoss = 0$. Links are provisioned to support the short-term burst write bandwidth:

$$links_{min} = \left\lceil \frac{avgUpdateRate \times burstMultiplier}{linkBW} \right\rceil$$

- *Write-order preserving asynchronous mirroring (async)*: this conservative protocol propagates all primary writes (without coalescing rewrites) to the secondary as fast as the inter-array interconnect allows. Updates are applied in the same order at both sites, but updates to the secondary may lag. This asynchrony can improve the performance of the foreground workload beyond inter-site distances of a few tens of km, but updates may be lost if the primary fails. We configure the primary with a write buffer that is large enough to smooth the observed worst-case update bursts for the workload. As a result, the links are provisioned to support the long-term average (non-unique) update rate:

$$links_{min} = \left\lceil \frac{avgUpdateRate}{linkBW} \right\rceil$$

  If the primary fails, then any updates buffered in the write buffer are lost. The worst-case data loss window is given by the time to fill or drain the buffer:

$$dataLoss = \frac{mirrorCacheCapacity \times numDiskArrays}{min(avgUpdateRate, (numLinks \times linkBW))}$$

- *Batched asynchronous mirroring with write absorption (asyncB)*: this protocol reduces bandwidth costs by coalescing repeated writes to the same data. Updates accumulate into batches at the primary and periodically propagate to the secondary mirror, which applies each update batch atomically [Patterson2002, Ji2003]. We declare batch boundaries at fixed time intervals ($interval_{asyncB}$) ranging from one minute to 24 hours. The link bandwidth must support the worst-case unique update rate over the batch interval:

$$links_{min} = \left\lceil \frac{uniqueUpdateRate(interval_{asyncB})}{linkBW} \right\rceil$$

  The potential data loss is the size of two delayed batches (one accumulating and one in transit to the secondary), so we approximate the worst-case loss window as twice the batch interval:

$$dataLoss = 2 \times interval_{asyncB}$$

The overall annualized outlay cost for mirroring is just the cost of the secondary copy plus the links:

$$mirrorCost = secondaryCost + numLinks \times linkCost$$

$$secondaryCost = primaryCost$$

Failover to a remote mirror and active standby host resources causes a short, temporary outage (30 seconds in our model). Reconstructing the entire data set on the primary from the secondary across the network takes longer:

$$recoveryTime = wkldCapacity / (linkBW \times numLinks)$$

The time to restore an array's worth of data after an array failure is calculated in a similar fashion, but scaled by $numDiskArrays$.

Both failover and reconstruction alternatives may require spare storage and/or computational resources at the secondary site. These may impose additional costs and/or recovery delays (see Section 3.5).

3.4 Tape backup

Table 4 summarizes the parameters of the tape backup model. Backups occur at fixed intervals ranging from 4 to 48 hours. Periodic full backups are optionally interspersed with cumulative incremental backups, which copy only the data modified since the last full backup. For example, backup intervals of 24 hours with incremental cycle counts of 6, 13 or 27 days correspond roughly to weekly, bi-weekly, or monthly full backups interspersed with daily incremental backups.

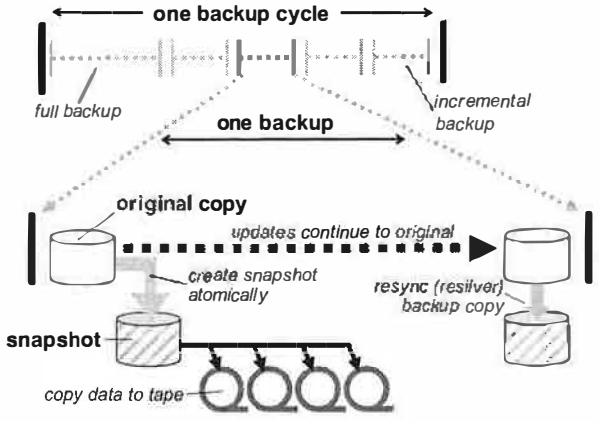


Figure 3: the backup cycle.

Parameter	Description	Values
tapeCapacity	tape media capacity (B)	SDLT: 320 GB LTO: 400 GB
tapeDriveBW	tape drive rate (B/sec)	SDLT: 16 MB/s LTO: 60 MB/s
$tapeDrives_{max}$	upper bound on number of drives in library	16
$tapes_{max}$	upper bound on number of tapes in library	600
$interval_{full}$	interval for full backups	4 hr, 12 hr, 24 hr, 48 hr
$interval_{incr}$	interval for incremental backups	4 hr, 12 hr, 24 hr, 48 hr
cycleCount	number of incrementals between full backups	0, 6, 13, 27
$interval_{cycle}$	interval between full backups	$= interval_{full} + cycleCount \cdot interval_{incr}$
$retrievalTime_{vault}$	time to retrieve tapes from offsite vault	1 hr
tapeLibraryCost	outlay cost for tape library, including chassis plus media slots	$148 342 per library
tapeDriveCost	outlay cost for tape drive	SDLT, LTO: $19 554 / drive
tapeCost	outlay cost for tape media cartridge	SDLT: $125 LTO: $150
fixedVaultCost	outlay for tape vault	$25 000 / year
vaultPer-ShipmentCost	outlay cost for a shipment to tape vault	$50 / shipment

Table 4: summary of tape backup parameters.

Figure 3 illustrates the backup process. It creates a consistent, read-only snapshot of the primary data, and then uses the snapshot as the source for the backup to tape (be it full or incremental). Snapshots may be taken using space-efficient copy-on-write techniques (e.g., [Patterson2002, Lee1996]), or by isolating a local mirror and synchronizing it with the primary copy after the backup is complete (e.g., [EMC–SRDF, HP–XP–CA]). The disk space required for a space-efficient incremental snapshot is determined from the average unique update rate and the backup interval.

Each backup must finish before the next one starts, effectively defining a *backup window* equal to the interval duration. The tool provisions sufficient tape drives to complete each backup within its window:

$$tapeDrives_{min} = \max(tapeDrives_{minFull}, tapeDrives_{minIncr})$$

where the number of drives needed for a full backup is:

$$tapeDrives_{minFull} = \left\lceil \frac{wkldCapacity}{interval_{full} \cdot tapeDriveBW} \right\rceil$$

and the number of tape drives required for the largest incremental backup is:

$$tapeDrives_{minIncr} = \left\lceil \frac{(cycleCount \quad 1) \cdot uniqueCapacity(interval_{incr})}{interval_{incr} \cdot tapeDriveBW} \right\rceil$$

Tapes are retained for a single full backup cycle, which includes the last full backup and all subsequent incremental backups. Each full backup is written onto a new set of tapes rather than the tapes for the previous full backup, in case it fails to complete. When a full backup completes, the tapes for the previous full backup are sent to the vault, and the tapes at the vault are recycled back to the primary site. The tapes are kept at the primary until this time in case they are needed quickly to respond to operator errors (e.g., rm *). Thus the total number of retained tapes is:

$$numTapes = 2 \times numTapes_{full} + numTapes_{incr}$$

where the number of tapes required for a full backup is:

$$numTapes_{full} = \lceil wkldCapacity / tapeCapacity \rceil$$

The number of tapes required for all incremental backups during a cycle is calculated by summing the number of tapes used for each one. We assume that each backup starts on a new tape. Taking into account the fact that the full backup interval may be larger than the incremental one, we get:

$$numTapes_{incr} = \sum_{i=0}^{cycleCount \quad 1} \left\lceil sizeOfIncr(i) / tapeCapacity \right\rceil$$

$$sizeOfIncr(i) = uniqueCapacity(interval_{full}) + i \cdot uniqueCapacity(interval_{incr})$$

Tape libraries hold both tape drives and tape cartridges. We model HP's series ESL9595 libraries, which can handle 2 to 16 drives, up to 600 cartridges, and three different tape cartridge/drive technologies (DLT, SDLT and LTO). The number of tape libraries needed to house the tapes and tape drives is:

$$numTapeLibraries = \max\left(\left\lceil \frac{numTapeDrives}{tapeDrives_{max}} \right\rceil, \left\lceil \frac{numTapes}{tapes_{max}} \right\rceil\right)$$

We assume that tapes are replaced every year, to guard against media failures. Thus, the minimum annualized outlay cost for backup is:

$$backupCost = \frac{\left(\begin{matrix} numTapeLibraries \cdot tapeLibraryCost + \\ numTapeDrives \cdot tapeDriveCost \end{matrix}\right)}{depreciationPeriod} + numTapes \cdot tapeCost$$

To protect against site disasters we add a second tape library at the reconstruction site. Disaster protection also incurs an annual cost for tape vaulting:

$$backupCost = 2 \cdot \frac{(numTapeLibraries \cdot tapeLibraryCost + numTapeDrives \cdot tapeDriveCost)}{depreciationPeriod} + numTapes \cdot tapeCost + fixedVaultCost + vaultPerShipmentCost \cdot shipmentsPerYear$$

3.4.1 Data loss

A primary array failure may destroy any backup in progress at the time of the failure, possibly losing all updates from both the current (accumulating) backup interval and the previous (propagating) backup interval. Assuming full intervals are at least as long as incremental intervals, the worst-case data loss window is the sum of the full and incremental backup intervals:

$$dataLoss = interval_{full} + interval_{incr}$$

In the event of a primary site disaster, the worst-case data loss occurs if the site is destroyed just before the new full backup completes and the old full backup is shipped offsite. In this case, the data at the vault is out-of-date by twice the full backup cycle duration, plus the interval for the latest full backup:

$$dataLoss = 2 \times interval_{cycle} + interval_{full}$$

3.4.2 Recovery time

Recovery from tape is a three-phase process: first, if the tapes are stored at an offsite vault, they must be retrieved to the reconstruction site; second, the latest full backup is restored and third, the latest subsequent incremental backup is restored. Vaults can be close to or far away from the target reconstruction site. The largest capacity incremental backup is the last one of the cycle. We make the following simplifying assumptions: all the tape drives in each library operate in parallel during each phase, and data is spread evenly across the tapes and drives. We ignore tape load time because it is typically less than 5% of the time to read the tape.

The number of tapes for the last incremental backup is:

$$numTapes_{maxIncr} = \left\lceil \frac{(cycleCount - 1) \cdot uniqueCapacity(interval_{incr})}{tapeCapacity} \right\rceil$$

For a site disaster, the worst-case recovery time is the time to retrieve the tapes from the offsite vault, plus the time to restore the last full and the last incremental backup of a cycle:

$$recoveryTime = retrievalTime_{vault} + recoveryTime_{full} + recoveryTime_{incr}$$

$$recoveryTime_{full} = \frac{tapeCapacity \cdot numTapes_{full}}{tapeDriveBW \cdot numTapeDrives}$$

$$recoveryTime_{incr} = \frac{tapeCapacity \cdot numTapes_{maxIncr}}{tapeDriveBW \cdot numTapeDrives}$$

Recovery after an array failure is analogous; the main differences are that the tapes are not yet offsite (e.g., zero $retrievalTime_{vault}$), and the tape counts for the full and last incremental backups are scaled by *numDiskArrays*.

3.5 Spare resources

After a failure, recovery can begin immediately if hot standby resources are available. Otherwise, resources must be found or acquired, drained if they are in use for another purpose, and (re)configured or (re)initialized if necessary. Solutions may minimize recovery time by keeping spare equipment in various states of readiness.

Failover or reconstruction at the secondary site requires computational resources at the target site. Since our focus is on storage system design, our tool always provisions hot spare server(s) at the target site if it selects one of these recovery alternatives. For simplicity we set the cost equal to the outlay cost of the primary disk arrays for the workload, following the rule of thumb that storage and servers each account for one third of the cost of a typical server site.

For reconstruction alternatives, we consider only reconstruction at the secondary site or at the primary over the same interconnect provisioned for backup or mirroring. Reconstruction at a third site would require additional, independently provisioned network links.

Reconstruction requires one or more disk arrays to serve as target for the restoration. We model a spectrum of spare resource options by the outlay cost of maintaining them and the time to provision and configure them. Table 5 gives the model parameters, and Figure 4 illustrates the alternatives. In all cases, we assume that any spare resources consumed by recovery are eventually replaced with new equipment, and factor the replacement equipment cost out of the equations.

Parameter	Description	Values
spareCost	outlay cost of spare disk array storage and facilities	*=primaryCost*
spareDiscount	discount factor for shared resources (fraction in [0,1])	0.2
t_{order}	time to order, deliver, and set up new resources	24 hr
$t_{identify}$	time to identify that resources are available	60 sec
$t_{configure}$	time to configure resources	10 hr
t_{scrub}	time to scrub resources	5 hr
$t_{negotiate}$	time to negotiate the release of shared resources	4 hr

Table 5: model parameters for spare resources. Spare costs include the cost of the entire target site for site disasters; for array failures, costs include only the array and related capacity-dependent facilities costs.

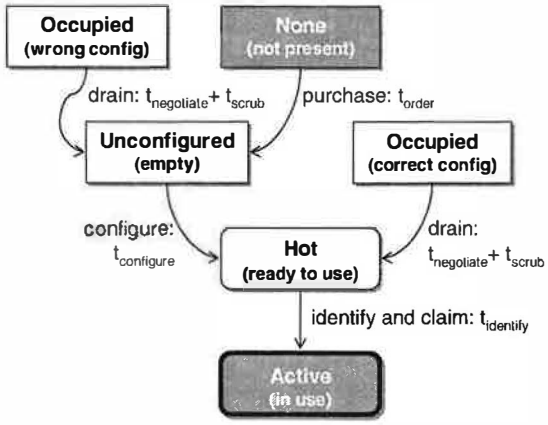


Figure 4: spare resource alternatives, and their preparation times. The time to provision spare resources is calculated by summing the times along the path from the chosen alternative to the active state.

One way of gaining access to spare resources is to rent access to a shared resource pool. Several companies offer such a service, which can be much cheaper than a dedicated backup site. We model the cost of shared resources by a fixed discount factor (a fraction in [0, 1]). However, the lower cost comes at a risk:

> *"During the days following September 11, some disaster recovery vendors found they were unable to accommodate all of their affected clients, with the result that several institutions found themselves without the anticipated backup facilities."* *[SEC2002]*

4 Automating dependability design

Our automated design tool, or *solver*, uses the data protection and recovery models described in the previous section to determine the most cost-effective solution for a given set of workload inputs, component parameters, assumed rates of primary array failures and primary site disasters, and failure penalty rates. Outputs from the solver include the choice of design alternative, the worst-case data loss window and recovery time (RPO and RTO), and the outlays and expected penalties. The solution also specifies settings for configuration parameters such as the number of tape drives, backup intervals, update batch interval, and other model parameters discussed in Section 3.

4.1 The optimization problem

We based our solver on mathematical programming primarily because of its expressive power and efficiency in traversing a large and complex search space. The technique has the added advantage that it arrives at a mathematically optimal solution; however, because the failure penalty rates are merely estimates, this fact is less important. Because our emphasis is on storage system design and not optimization technology, we used an off-the-shelf optimization tool, rather than develop our own. We cast the problem as a mixed-integer optimization with constraints. Our prototype is written in the GAMS language for the CPLEX solver [GAMS1998, ILOG2002].

The solver's objective is to minimize overall business cost, defined as outlays plus failure penalties. Within the solver, a set of binary decision variables represents the data protection alternatives and their base configurations. Each binary variable corresponds to a single protection alternative (e.g., mirroring or backup) and a specific set of discrete configuration parameters (e.g., "batched asynchronous mirroring with a write absorption interval of one minute"). A second set of binary decision variables represents the alternatives for recovery and spare resources (e.g., "use failover"). Integer decision variables represent the number of bandwidth devices (e.g., network links or tape drives) for each protection alternative.

The solver uses the following optimization constraints:

- Exactly one protection alternative must be chosen.
- Exactly one recovery alternative must be chosen.
- The number of bandwidth devices for a data protection alternative is either zero (if that alternative has not been chosen), or it is within the range specified by the upper and lower bounds calculated or specified for the alternative (e.g., $[links_{min}, links_{max}]$).
- The aggregate device bandwidth may not exceed the aggregate reload rate for the primary disk array(s).

4.2 Optimizing the optimization

The multitude of parameter combinations leads to a complex search space. We applied several techniques to make the solver more efficient. With these improvements, the optimization runs in just a few seconds.

First, we reduced the search space considerably by quantizing the values of certain continuous parameters, such as backup intervals and the batch intervals for the batched asynchronous remote-mirroring scheme.

Second, we divided several continuous, non-linear functions to a set of piecewise-linear segments. When formulated in the straightforward fashion described above, the recovery time models have terms that depend inversely on the number of links or tape drives y. The resulting optimization problem becomes non-linear. Although solvers exist for certain classes of non-linear optimization problems, they may take an unacceptably long time to find a solution, or fail to find one at all. Linear solvers exploit well-known theoretical results about the search space structure to solve significantly

larger problems in seconds. To linearize our problem, we defined new variables z standing for the inverse of the problematic y terms. We introduced new constraints on the variables so that the resulting optimization problem is equivalent to the original one. Since the transformation is convex, we used the methods described in [Williams1999], which linearize convex constraints in continuous variables. If the variables had been continuous, this would have been an approximation, but because our decision variables are integers, we can prove that the result is exact.

5 Experimental results

We conducted a series of experiments to explore the behavior of our automated dependability design tool:

1. To validate the solver's recovery time, data loss and cost calculations, we evaluated these quantities for a fixed set of designs.
2. To explore the solver's design choices, we supplied a set of penalty inputs corresponding to specific industry segments, and examined the output designs.
3. To explore the sensitivity of the solver's design choices to its inputs, we varied the penalty rates across a wide range, and examined the outputs.

Although workload parameters do affect the configuration parameters for each solution, our experience has shown that the penalty rates dominate the design choices. We use a single workload for all experiments (cello2002, described in Table 1). All experiments assume a failure rate of one primary site disaster per year. The effect of changing the failure frequency is equivalent to scaling the penalty rate values by an appropriate multiplier; the experiments explore in detail how the design choices change as these penalty rates vary.

5.1 Validation of fixed designs

To validate our intuitions about the recovery time, data loss and cost behaviors of candidate designs, we ran the solver in *evaluation* mode. In this mode, the solver's decision variables are fixed to a particular design choice and configuration parameter values. We present a representative sampling of the results here, all of which were set up to reconstruct the data to a hot spare at the primary site in the event of a site disaster.

Figure 5 presents a set of results for tape backup. The graph shows recovery time as a function of the number of tape drives for two different tape drive technologies (SDLT and LTO). We considered two backup schedules: one using just full backups with a four-hour backup interval, and one using weekly full backups plus daily incremental ones, both with 24-hour intervals. The graphs for the full-only policy begin at two drives (for LTO) and seven drives (for SDLT) because that is the minimum number of tape drives required to back up the entire data set in the four-hour window. For LTO, the curves stop at eight drives because the aggregate tape drive bandwidth beyond this point is larger than the array reload rate, so additional drives do not help.

As expected, the slower SDLT technology results in longer recovery times. For both technologies, recovery is faster with the full-only schedule: it restores only the last full backup, while the weekly full/daily incremental schedule must also apply the latest incremental backup.

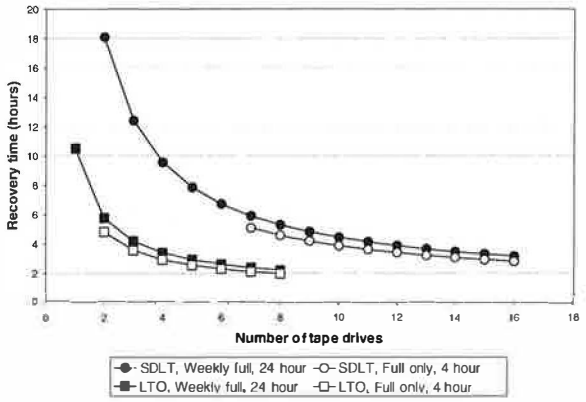


Figure 5: tape backup recovery time as a function of tape drive/cartridge technology and backup policy.

The graph confirms the importance of provisioning to minimize recovery time in this scenario: for smaller numbers of tape drives, the outage penalty reduction from adding a tape drive exceeds the drive's purchase price. The LTO drives are particularly cost-effective.

Protection technique	Recent data loss
Synchronous mirroring	0
Asynchronous mirroring	2.2 min
Asynchronous batch mirroring (interval = 1 min)	2.0 min
Asynchronous batch mirroring (interval = 1 hour)	2.0 hours
Backup (full-only; interval = 4 hours)	12 hours
Backup (weekly full + daily incremental; both intervals = 24 hours)	360 hours

Table 6: recent data loss for different designs.

Table 6 summarizes the worst-case data loss for several design alternatives. Synchronous mirroring experiences zero data loss. For non-coalescing asynchronous mirroring, data loss corresponds to the time for the workload to fill the 100 MB mirroring write cache. Data loss for the asynchronous batch variants is twice the batch duration. The worst-case data loss for tape backup and

vaulting is twice the backup cycle, plus the backup interval for the latest full backup. The large values show the down-side of keeping tapes on-site rather than sending them to the vault in the case of site failure.

Figure 6 shows how outlay and expected penalty costs combine to yield the total cost of a candidate solution across a range of configuration parameters. The graph shows outlay and penalty costs for remote asynchronous mirroring with recovery by reconstruction at the primary. This experiment assumes the penalty rate inputs for both data loss and outages are $20 000 per hour.

Outlay costs increase linearly as the number of configured T3 network links increases. The expected data loss for each failure is constant with this solution, but adding more links speeds recovery by increasing the bandwidth from the mirror to the primary. The graph shows a "sweet-spot" that minimizes the overall cost at five T3 links: with fewer links, slower recovery drives up the outage penalties, while adding more links yields diminishing benefit and causes link outlays to dominate. We ran this experiment in evaluation mode for illustrative purposes; in design mode the goal of the solver is to find such sweet spots automatically.

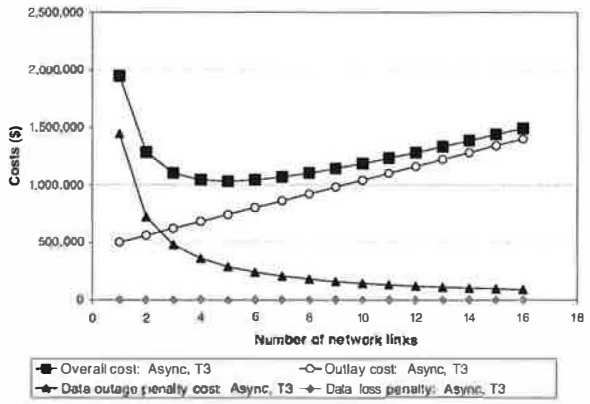


Figure 6: outlay and penalty costs for asynchronous mirroring as a function of link configuration.

5.2 Automated design choices

For the second set of experiments we ran the tool in design mode to configure a data protection system for one primary site disaster per year. These experiments explore the design choices for penalty rates that are typical of specific industry segments [CNT2003b]. Table 7 summarizes the data loss and outage penalty inputs for each industry segment, together with the chosen design and its cost. Figure 7 illustrates the cost breakdown for each solution.

Type	Data loss penalty	Outage penalty	Design chosen	Overall cost
Student accounts: Storage of data owned by students is tolerant of data loss and outages.				
	$500 / hr	$500 / hr	backup with 12-hr interval + no spares	$301k
Company documents: Documents such as papers, presentations, and design documents are tolerant to small outages and loss of recent writes.				
	$500k / hr	$500 / hr	async mirror+ reconstruction + no spares	$426k
Web server for an online retailer: Outages are expensive, because if the web server goes down, orders stop. Data loss is tolerable, because data can be replaced from other sources.				
	$500 / hr	$500k / hr	asyncB mirror + failover	$506k
Consumer banking: Consumer banking services are tolerant of modest outages (since account holders are unlikely to switch banks), but not data loss.				
	$50M / hr	$50k / hr	sync mirror + failover	$562k
Central bank: Central banks are required by regulations to have zero data loss and small outage windows.				
	$50M / hr	$5M / hr	sync mirror + failover	$603k

Table 7: summary of industry segment examples.

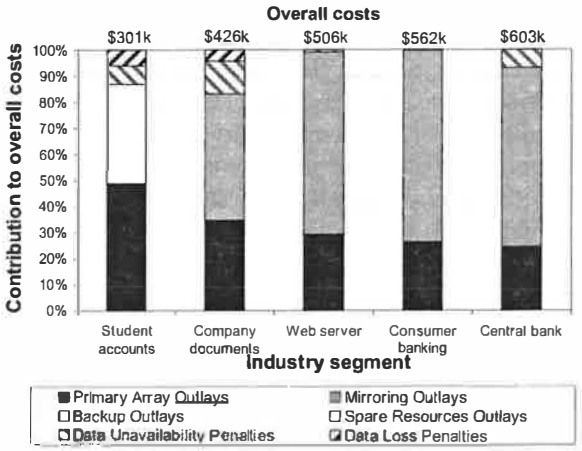


Figure 7: solution costs by industry segment.

For lower penalty rates (student accounts and company documents), the tool chooses designs that employ reconstruction with no spare resources: the outlay costs for better protection exceed the failure penalties. It switches to asynchronous mirroring and failover at moderate penalty rates (web server), because the outlays for this stronger solution are less than the penalties expected with a cheaper solution. At the highest penalty rates (consumer banking and central bank), the tool selects an aggressive synchronous mirroring + failover

solution to achieve zero data loss and minimal outage time. Even so, the mission-critical central bank incurs a non-trivial outage penalty for the 30-second failover time.

Interestingly, the designer always chooses a solution whose expected penalties are a modest share of the overall cost (at most 17%).

5.3 Solver design choice sensitivity

Our final set of experiments maps the solution space for data protection and recovery across a range of failure penalty rates. We performed over 500 separate experiments, varying the data loss and data outage penalty rates independently over a wide range (10^7:10^1). Exploring this space by hand using existing tools would be unthinkable. Our intent is to give insight into the richness of the design space, to show the feasibility of navigating this design space automatically, and to provide further evidence that the design tool is correct and efficient.

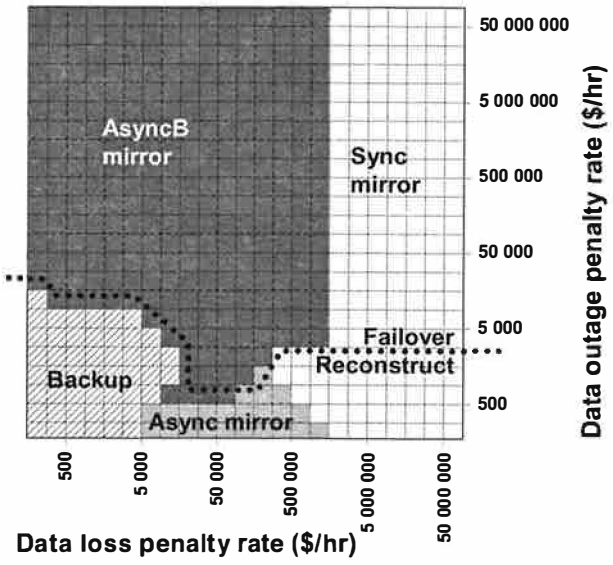


Figure 8: solution types picked by the solver for the space of penalty rates. The dotted line separates failover and reconstruction solutions.

Figure 8 shows the design alternatives selected for each input scenario. As expected, the tool chooses the cheapest solution—tape backup—for scenarios tolerant of outages and data loss (e.g., student accounts), but incorporates hot spare resources and more aggressive backup intervals as the penalties for outage and data loss increase. The tool chooses asynchronous batch mirroring for the scenarios with more severe outage penalties when the cost to lose recent writes is acceptable (e.g., web server). At low data outage penalty rates, as the penalty rate for data loss continues to increase (e.g., company documents), the solutions switch to order-preserving asynchronous mirroring to reduce the data loss beyond what tape backup can offer. As the penalty rates for data loss increase further (e.g., consumer banking and central bank), the solutions shift to synchronous mirroring.

When outage penalties are low, the mirroring solutions use reconstruction rather than failover to avoid incurring the additional outlays for standby compute resources. As outage penalties increase, the solutions incorporate hot spare standby resources and failover to minimize downtime.

To clarify the reasoning behind the solver's decisions, we now look at its calculations in greater detail. Figure 9 and Figure 10 show the recovery time and recent data loss, respectively, for the chosen solutions.

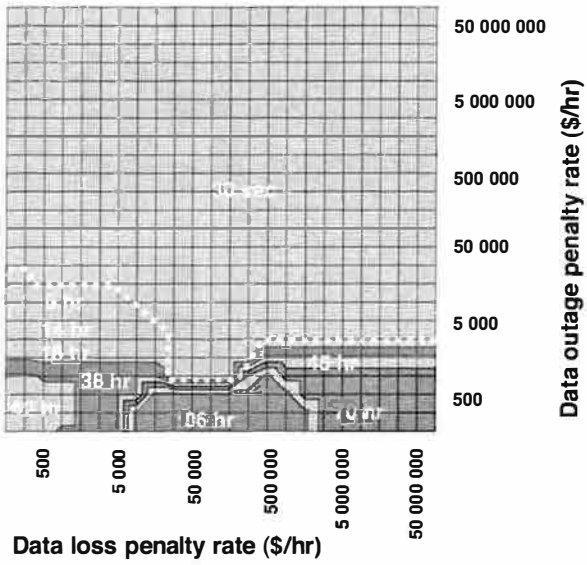


Figure 9: recovery time as a function of penalty rates. Each tinted band corresponds to a 20–hour increase in recovery time. The dotted line separates failover and reconstruction solutions.

For the lowest data loss and outage penalty rates, the solver chooses tape backup with no spare resources, full-only backups and twelve-hour intervals, resulting in a recovery time of 42 hours and data loss of 36 hours. As the data outage penalty increases, the backup-based solutions shift to using occupied spare resources and four-hour backup intervals, decreasing recovery time to 14 hours. The reductions in backup intervals also reduce the data loss window to 12 hours. Adding hot standby resources further reduces recovery time to five hours. The shift to asynchronous batch mirroring + failover at high outage penalty rates dramatically reduces the recovery time to 30 seconds and the data loss window to two minutes.

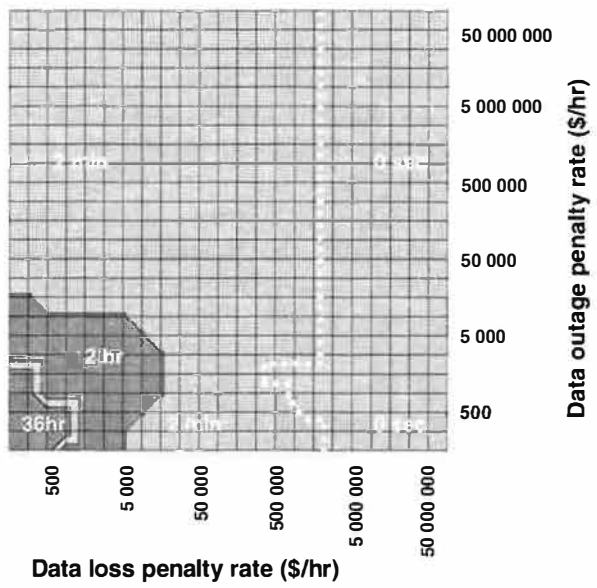


Figure 10: data loss time as a function of penalty rates. Each tinted band corresponds to a 10-hour increase in data loss time. The dotted line separates asynchronous and synchronous mirroring.

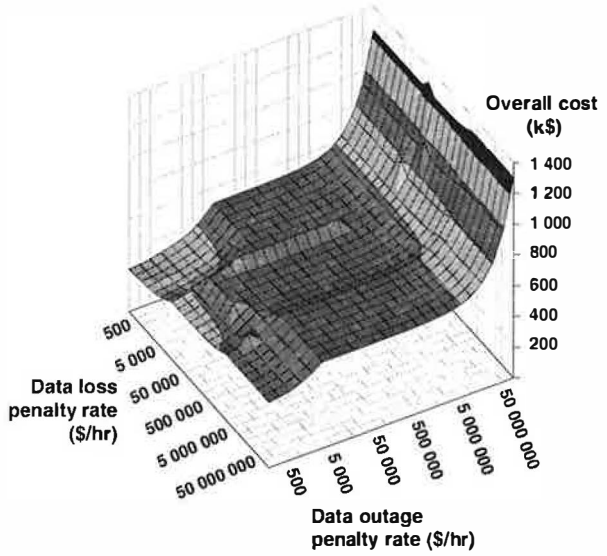


Figure 11: total (penalty+outlay) costs (k$) for the space of penalty rates.

For asynchronous mirroring with low outage penalties, the tool selects reconstruction rather than failover; it tolerates 100+ hour outage times, rather than incur the expense of standby compute resources for failover. As the loss penalty rate increases, the shift to synchronous mirroring + reconstruction reduces data loss to zero, with the added benefit of decreasing the recovery time to about 70 hours. (This reduction comes from the additional networking links synchronous mirroring requires to support update bursts.) As the outage penalty rates increase, the synchronous mirroring solutions shift to failover, resulting in recovery times of 30 seconds.

Figure 11 shows the overall costs associated with the solver's choices. In all cases the tool balances outlays with penalties, increasing system cost only as needed to avoid larger penalties. One notable crossover occurs as more aggressive spare resource options are purchased (from *none* to *occupied* to *hot spare* to *failover*), to reduce the outage penalties. Another interesting crossover occurs as the choice of protection technique shifts (from more aggressive backup policies to asynchronous mirroring to synchronous mirroring) to reduce data loss penalties.

5.4 Discussion

Asynchronous mirroring performed surprisingly well in this study. We were initially skeptical, but found that the additional high costs of high-speed, long distance links largely relegated synchronous mirroring to situations where the small data loss that could result with asynchronous mirroring is incredibly expensive – as it is, for example, in the central bank case, where a single transfer can involve millions or even billions of dollars.

We noted that the solver generally chose to use full-only backups, rather than interspersing incremental backups between full ones. Even though the tape costs for incremental backups are generally lower, this outlay savings is outweighed by the increased recovery time penalties that result from the need to restore both the latest full and latest incremental backups.

We observed firsthand that even people who should know better can make mistakes when faced with this rich design space. For example, we were experimenting with an early version of the design tool, exploring the effects of increasing the outage penalty rate for a mirroring + reconstruction solution to protect against array failure, when we noticed that the tool chose to configure a local tape library at the primary site. Why? Because we had constrained mirroring in that version to provision only enough networks links to absorb the update traffic for asynchronous mirroring. Unfortunately, the time to reconstruct the primary over these economical, slow links was so large that it became cost-effective to restore from a local tape library, rather than trickle the data back from the remote mirror. The original design was wrong: the solution should have been designed to handle the recovery case, not just the mirror update case. We should not have made this mistake, but we did. The automated dependability design tool can help to avoid this type of mistake.

6 Related work

We found very few other systems and tools that automate the design of dependability solutions. Most of the tools we found tackle only pieces of the problem, and rely on people to do the rest.

Storage vendors, such as IBM, Sun, HP, Computer Associates and Exabyte, provide web-based and downloadable tools to help with low-level backup provisioning questions, and to estimate total cost of ownership (TCO) and return on IT investments (ROI). For example, Sun's backup calculator [Sun2004] computes the number of libraries, tape drives and tape cartridges required to support a backup schedule whose parameters are supplied as inputs. This type of tool provides no indication of the dependability of the resulting backup system, nor any indication of the financial ramifications of its use.

ROI and TCO calculators (e.g., [EMC2004]) provide estimates for the total cost of ownership for alternative storage systems, given inputs for storage capacity, utilization and availability. These tools incorporate outlay costs for storage and server hardware and administrator salaries, and penalty costs for outages (but not data loss). Their overall system costs are based on input values for system availability, rather than calculated values for the dependability of a storage system configuration.

These tools provide simplistic evaluations of low-level configuration parameters or overall system costs, respectively. By changing input parameters, one can perform a rudimentary sensitivity analysis. Neither class of tools can propose a storage system configuration to meet user-specified dependability goals.

Like many other researchers in the dependability community, we build models of the systems we are trying to design. Our work emphasizes embedding those models in an automated design system. Most other work, however, focuses on models and modeling toolkits that allow designers to interactively try "what-if" analyses by hand (e.g., [Deavours2002, Haverkort2001]).

System designers and analysts make the design choices we describe here every day, but the focus in their literature is on operational issues, such as determining backup policies and setting related configuration parameters (e.g., [Cougias2003, Marcus2003, Preston1998, daSilva1993]).

The majority of "data protection" work is focused on developing new protection mechanisms. Only rarely do their designers consider the difficulties of deciding when to use them, and how they combine with existing techniques [Wylie2001].

Researchers have developed automated tools to design systems to meet performance goals (e.g., [Anderson2002a, Anderson2002b, Alvarez2001, Borowsky1997]), but they considered only online redundancy techniques (mirroring and RAID 5). Our dependability design work builds on the experience gained there.

We also leverage existing work that deals with specifying and evaluating dependability requirements. Examples include a declarative method of specifying data performability and reliability requirements [Keeton2002, Wilkes2001], and ways to measure the availability of RAID systems [Brown2000].

Several recent papers from CNT (http://www.cnt.com) are particularly helpful in providing an overview of the growing regulatory and legal pressures on disaster tolerance in the financial industry. They provide some information about outage penalties, but little on the cost of lost data.

7 Conclusions

Data *must* be dependable: the cost of losing it, or even of losing access to it, is simply too high. The complexity of the systems to achieve this goal, and the range of failure cases they must tolerate, is increasing rapidly. What's more, human designers don't always make the right choices, or understand the ramifications of the choices they do make.

The solution is to automate the design of data dependability solutions. Our design tool provides numerous benefits over the manual approach employed today. By treating the question as an optimization problem, it can search the entire design space and find the best alternative for a set of business requirement inputs. Since the process is automated, it is easy (and fast) to evaluate a broad range of inputs to support "what if" analyses. Furthermore, the tool doesn't just provide "black box" answers, but can provide insights into the choices it makes, so that customers can understand and interpret its decisions.

Opportunities exist to extend this work in several different directions: improving the quality of the data protection technique models; enhancing the financial exposure model; extending the scope of the solutions and failures imposed; making it easier to add new data protection and recovery techniques to an existing solver; and generating easy-to-use visual interfaces to the tool itself. We hope to explore many of these avenues.

More details can be found in an extended version of this paper [Keeton2004].

Acknowledgements

Thanks to Minwen Ji and Alistair Veitch for helping us understand the intricacies of remote mirroring, Fereydoon Safai for helping with our test infrastructure, and Christos Karamanolis, Arif Merchant and our shepherd, Chandu Thekkath, for their helpful feedback on earlier versions of this paper.

References

[Alvarez2001] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, and J. Wilkes. *Minerva: an automated resource provisioning tool for large-scale storage systems.* ACM Transactions on Computer Systems **19**(4):483–518, November 2001.

[Anderson2002a] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. *Hippodrome: running circles around storage administration.* Proc. File and Storage Technologies (FAST), pp. 175–188, January 2002.

[Anderson2002b] E. Anderson, R. Swaminathan, A. Veitch, G. Alvarez, and J. Wilkes. *Selecting RAID levels for disk arrays.* Proc. File and Storage Technologies (FAST), pp. 189–201, January 2002.

[Borowsky1997] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. *Using attribute-managed storage to achieve QoS.* 5th International Workshop on Quality of Service (IWQoS'97), June 1997. Available from http://www.hpl.hp.com/SSP/papers.

[Brown2000] A. Brown and D. Patterson. *Towards availability benchmarks: a case study of software RAID systems.* Proc. 2000 USENIX Annual Technical Conference, pp. 263–276, June 2000.

[CNT2003] *Legal requirements for data retention and recovery in the financial industry.* Report PL731. Computer Network Technology Corporation. Sep. 2003. http://www.cnt.com/literature/documents/PL731.pdf, accessed Oct. 2003.

[CNT2003b] *Disk mirroring -- local or remote.* Report PL512, Computer Network Technology Corporation, August 2003. http://www.cnt.com/literature/documents/pl512.pdf, accessed Oct. 2003.

[Cougias2003] D. Cougias, E. L. Heiberger, K. Koop. *The Backup Book: Disaster Recovery from Desktop to Data Center.* Network Frontiers, LLC, 2003.

[daSilva1993] J. daSilva and O. Gudmundsson. *The Amanda network backup manager.* Proc. 7th USENIX Systems Administration Conference (LISA), pp. 171-182, Nov. 1993.

[Deavours2002] D. D. Deavours, et al. *The Möbius Framework and its implementation.* IEEE Transactions on Software Engineering, **28**(10):956–969, October 2002.

[EagleRock2001] *Online survey results: 2001 cost of downtime.* Eagle Rock Alliance Ltd, Aug. 2001. http://contingencyplanningresearch.com/2001 Survey.pdf accessed May 2003.

[EMC2004] EMC Networked Storage ROI & TCO Calculator. http://www.emc.com/forms/commercial/roi_calculator/project.jsp accessed January 2004.

[EMC–SRDF] *Symmetrix Remote Data Facility product description guide.* June 2000. EMC Corporation.

[GAMS1998] A. Brooke *et al. GAMS: a user's guide.* GAMS Development Corp.: Washington, DC. 1998.

[Haverkort2001] B. Haverkort, R. Marie, G. Rubino and K. Trivedi, eds. *Performability modeling: techniques and tools,* John Wiley & Sons, May 2001.

[HP–XP–CA] *hp continuous access xp.* Product brief 5980–8608EN, 2001. Hewlett-Packard Company.

[ILOG2002] *CPLEX 8.0 user's manual.* Ilog, Inc: Mountain View, CA. July 2002.

[Ji2003] Minwen Ji, Alistair Veitch, and John Wilkes. *Seneca: remote mirroring done write.* USENIX Technical Conference (USENIX'03) pp. 253–268. June 2003.

[Keeton2002] K. Keeton and J. Wilkes. *Automating data dependability.* Proc. 10th ACM-SIGOPS European Workshop, pp. 93–100, Sept. 2002 (Saint-Emilion, France).

[Keeton2004] K. Keeton, C. Santos, D. Beyer, J. Chase and J. Wilkes. *Designing for disasters – extended version.* Hewlett-Packard Laboratories Technical Report HPL-2004-16, available from http://www.hpl.hp.com/SSP/papers.

[Lee1996] Edward K. Lee and Chandramohan A. Thekkath. *Petal: distributed virtual disks.* Proceedings of ASPLOS VII. Published as *SIGPLAN Notices* **31**(9):84-92. Oct. 1996.

[Marcus2003] E. Marcus and H. Stern. *Blueprints for High Availability.* Wiley Publishing, Inc., 2003.

[Patterson2002] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. *SnapMirror: file-system-based asynchronous mirroring for disaster recovery.* Proc. File and Storage Technologies (FAST), pp. 117–129, Jan. 2002.

[Preston1998] W. C. Preston. *Using Gigabit Ethernet to backup six terabytes.* Proc. 12th USENIX Systems Administration Conference (LISA), pp. 87-96, Dec. 1998.

[SEC2002] *Summary of "lessons learned" from events of September 11 and implications for business continuity.* US Securities and Exchange Commission: Washington, DC. February 2002. http://www.sec.gov/divisions/marketreg/lessonslearned.htm

[Sun2004] Sun Microsystems Storage and Backup Requirements Calculator. http://www.sun.com/storage/tape/tape_lib_calculator.html, accessed January 2004.

[Wilkes2001] J. Wilkes. *Traveling to Rome: QoS specifications for automated storage system management.* Proc. the International Workshop on Quality of Service (IWQoS), pp. 75–91, June 2001.

[Williams1999] H. P. Williams. *Model building in mathematical programming.* John Wiley & Sons: New York. 4th edition, 1999.

[Wylie2001] J. Wylie, et al. *Selecting the right data distribution scheme for a survivable storage system,* Technical report CMU-CS-01-120, Carnegie Mellon University, May 2001.

Diamond: A Storage Architecture for Early Discard in Interactive Search

Larry Huston, Rahul Sukthankar, • Rajiv Wickremesinghe, M. Satyanarayanan, •
Gregory R. Ganger,• Erik Riedel,* Anastassia Ailamaki•
*Intel Research Pittsburgh, •Carnegie Mellon University, Duke University, *Seagate Research*

Abstract

This paper explores the concept of *early discard* for interactive search of unindexed data. Processing data inside storage devices using downloaded *searchlet* code enables Diamond to perform ef cient, application-speci c ltering of large data collections. Early discard helps users who are looking for needles in a haystack by eliminating the bulk of the irrelevant items as early as possible. A searchlet consists of a set of application-generated lters that Diamond uses to determine whether an object may be of interest to the user. The system optimizes the evaluation order of the lters based on run-time measurements of each lter's selectivity and computational cost. Diamond can also dynamically partition computation between the storage devices and the host computer to adjust for changes in hardware and network conditions. Performance numbers show that Diamond dynamically adapts to a query and to run-time system state. An informal user study of an image retrieval application supports our belief that early discard signi cantly improves the quality of interactive searches.

1 Introduction

How does one nd a few desired items in many terabytes or petabytes of complex and loosely-structured data such as digital photographs, video streams, CAT scans, architectural drawings, or USGS maps? If the data has already been indexed for the query being posed, the problem is easy. Unfortunately, a suitable index is often not available and a user has no choice but to perform an exhaustive search over the entire volume of data. Although attributes such as the author, date, or other context of data items can restrict the search space, the user is still left with an enormous number of items to examine. Today, scanning such a large volume of data is so slow that it is only performed in the context of well-planned data mining. This is typically a batch job that runs overnight and is only rarely attempted interactively [15].

Our goal is to enable interactive search of non-indexed data, where the user wishes to retrieve a small set of important items buried in a large collection. For instance, consider a surveillance scenario where an analyst is monitoring satellite imagery for interesting activity around oil tankers. Current image processing algorithms may be able to automatically discard images that do not contain oil tankers, but they cannot detect interesting activity . Filtering the data allows the analyst to focus attention on the promising candidates by signi cantly reducing the number of irrelevant items. To make such systems practical, new techniques for scanning large volumes of data are needed. We believe that the solution lies in *early discard*, the ability to discard irrelevant data items as quickly and ef ciently as possible (*e.g.*, at the storage device rather than close to the user). We have developed a storage architecture and programming model called *Diamond* that embodies early discard. Diamond has been designed to run on an active disk [1, 20, 25] platform, but does not depend on the availability of active storage devices. It can be realized using diverse storage back ends ranging from emulated active disks on a general-purpose cluster to storage nodes on a wide-area network.

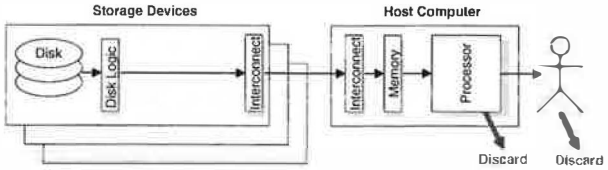
This paper focuses on *pure brute-force* interactive search (*i.e.*, where all of the data is processed for each query). Studying this extreme case enables us to determine the feasibility of early discard in a worst-case setting. Future Diamond implementations could incorporate performance optimizations such as caching results from previous queries and exploiting indices to reduce the search time.

This paper is organized as follows. Section 2 introduces early discard. Section 3 presents the Diamond architecture. Section 4 describes a proof-of-concept application and an informal user study. Section 5 discusses implementation details. Section 6 presents experimental results. Section 7 summarizes related work. Finally, Section 8 concludes the paper.

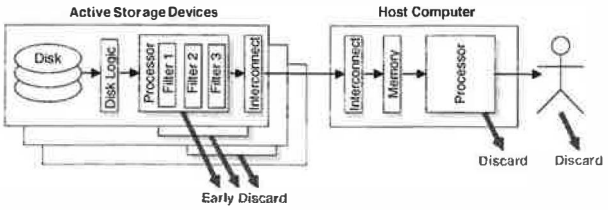
2 Background and Motivation

2.1 Limitations of Indexing

The standard approach to ef cient interactive search is to create an of ine index of the data. Indexing assumes that the mapping between the user's query and the relevant data can be pre-computed, enabling the system to ef ciently organize data so that only a small fraction is accessed during a particular search. Unfortunately, indexing complex data remains a challenging problem for several reasons. First, manual indexing is often infeasible for large datasets and automated methods for extract-



(a) Traditional storage architecture



(b) Diamond architecture

Figure 1: **Early Discard** - Unlike traditional architectures for exhaustive search, where all of the data must be shipped from to the host computer, the Diamond architecture employs early discard to efficiently reject the bulk of the irrelevant data at the active storage device.

ing the semantic content from many data types are still rather primitive (the *semantic gap* [23]). Second, the richness of the data often requires a high-dimensional representation that is not amenable to efficient indexing (a consequence of the *curse of dimensionality* [6, 9, 33]). Third, realistic user queries can be very sophisticated, requiring a great deal of domain knowledge that is often not available to the system for optimization. Fourth, expressing the user's needs in a usable form can be extremely difficult (*e.g.*, "I need a photo of an energetic puppy playing with a happy toddler"). All of these problems limit the usability of *interactive data analysis* [15] today.

2.2 Importance of Early Discard

Figure 1(a) shows the traditional architecture for exhaustive search. Each data item passes through a pipeline from the disk surface, through the disk logic, over an interconnect to the host computer's memory. The search application can reject some of the data before presenting the rest to the user. Two problems with this design are: (1) the system is unable to take full advantage of the parallelism at the storage devices; (2) although the user is only interested in a small fraction of the data, all of it must be shipped from the storage devices to the host machine, and the bulk of the data is then discarded in the final stages of the pipeline. This is undesirable because the irrelevant data will often overload the interconnect or host processor.

Early discard is the idea of rejecting irrelevant data as early in the pipeline as possible. For instance, by exploiting active storage devices, one could eliminate a large fraction of the data before it was sent over the interconnect, as shown in Figure 1(b). Unfortunately, the storage device cannot determine the set of irrelevant objects *a priori* — the knowledge needed to recognize the useful data is only available to the search application (and the user). However, if one could imbue some of the earlier stages of the pipeline with a portion of the intelligence of the application (or the user), exhaustive search would become much more efficient. This is supported by our experiments, as described in Section 6.3.

For most real-world applications, the sophistication of user queries outpaces the development of algorithms that can understand complex, domain-dependent data. For instance, in a homeland security context, state-of-the-art algorithms can reliably discard images that do not contain human faces. However, face recognition software has not advanced to the point where it can reliably recognize photos of particular individuals. Thus, we believe that a large fraction of exhaustive search tasks will be interactive in nature. Unlike a typical web search, an interactive brute-force search through a large dataset could demand hours (rather than seconds) of focused attention from the user. For example, a biochemist might be willing to spend an afternoon in interactive exploration seeking a protein matching a new hypothesis. It is important for such applications to consider the human as the most important stage in the pipeline. Effective management of the user's limited bandwidth becomes crucial as the size and complexity of the data grows. Early discard enables the system to eliminate clearly useless data items early in the pipeline. The scarcest resource, *human attention*, can be directed at the most promising data items.

Ideally, early discard would reject all of the irrelevant data at the storage device without eliminating any of the desired data. This is impossible in practice for two reasons. First, the amount of computation available at the storage device may be insufficient to perform all of the necessary (potentially expensive) application-specific computations. Second, there is a fundamental trade-off [9] between false-positives (irrelevant data that is not rejected) and false-negatives (good data that is incorrectly discarded). Early discard algorithms can be tuned to favor one at the expense of the other, and different domain applications will make different trade-offs. For instance, an advertising agency searching a large collection of images may wish to quickly find a photo that matches a particular theme and may choose aggressive filtering; conversely, a homeland security analyst might wish to reduce the chance of accidentally losing a relevant object and would use more conservative filters (and accept the price of increased manual scanning). It is important to note that early discard does not, by itself, impact the accuracy of the search application: it simply makes applications that filter data more efficient.

2.3 Self-Tuning for Hardware Evolution

The idea of performing specialized computation close to the data is not a new concept. Database machines [7, 17] advocated the use of specialized processors for efficient data processing. Although these ideas had significant technical merit, they failed, at the time, because designing specialized processors that could keep pace with the sustained increase in general-purpose processor speed was commercially impractical.

More recently, the idea of an *active disk* [1, 20, 25], where a storage device is coupled with a general-purpose processor, has become popular. The flexibility provided by active disks is well-suited to early discard; an active disk platform could run filtering algorithms for a variety of search domains, and could support applications that dynamically adapt the balance of computation between storage and host as the location of the search bottleneck changes [2]. Over time, due to hardware upgrades, the balance of processing power between the host computer and storage system may shift. In general, a system should expect a heterogeneous composition of computational capabilities among the storage devices as newer devices may have more powerful processors or more memory. The more capable devices could execute more demanding early discard algorithms, and the partitioning of computation between the devices and the host computer should be managed automatically. Analogously, when the interconnect infrastructure or host computer is upgraded, one may expect computation to shift away from the storage devices. To be successful over the long term, the design needs to be *self-tuning*; manual re-tuning for each hardware change is impractical.

In practice, the best partitioning will depend on the characteristics of the processors, their load, the type and distribution of the data, and the query. For example, if the user were to search a collection of holiday pictures for snowboarding photos, these might be clustered together on a small fraction of devices, creating hotspots in the system.

Diamond provides two mechanisms to support these diverse storage system configurations. The first allows an application to generate specialized early discard code that matches each storage device's capabilities. The second enables the Diamond system to dynamically adapt the evaluation of early discard code, and is the focus of this paper. In particular, we explore two aspects of early discard: (1) adaptive partitioning of computation between the storage devices and the host computer based on run-time measurements; (2) dynamic ordering of search terms to minimize the total computation time.

2.4 Exploiting the Structure of Search

Diamond exploits several simplifications inherent to the search domain. First, search tasks only require read access to data, allowing Diamond to avoid locking complexities and to ignore some security issues. Second, search tasks typically permit stored objects to be examined in any order. This order-independence offers several benefits: easy parallelization of early discard within and across storage devices, significant flexibility in scheduling data reads, and simplified migration of computation between the active storage devices and host computer. Third, most search tasks do not require maintaining state between objects. This "stateless" property supports efficient parallelization of early discard, and simplifies the run-time migration of computation between active storage device and host computer.

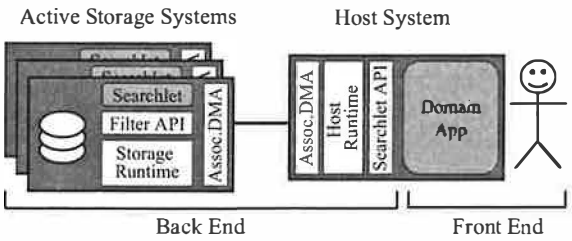


Figure 2: **Diamond Architecture**

3 Diamond Architecture

Figure 2 illustrates the Diamond storage architecture. Diamond provides a clear separation between the *front end*, which encapsulates domain-specific application code on the host computer, and the *back end*, which consists of a domain-independent infrastructure that is common across a wide range of search applications.

Diamond applications aim to reduce the load on the user by eliminating irrelevant data using domain-specific knowledge. Query formulation is domain-specific and is handled by the search application at the front end. Once a search has been formulated, the application translates the query into a set of machine executable tasks (termed a *searchlet*) for eliminating data, and passes these to the back end. The searchlet contains all of the domain-specific knowledge needed for early discard, and is a proxy of the application (and of the user) that can execute within the back end.

Searchlets are transmitted to the back end through the *searchlet API*, and distributed to the storage devices. At each storage device, the runtime system iterates through the objects on the disk (in a system-determined order) and evaluates the searchlet. The searchlet consists of a set of filters (see Section 5), each of which can independently discard objects. Objects that pass through all filters in a searchlet are deemed interesting, and made available to the domain application through the searchlet API.

The domain application may perform further processing on the interesting objects to see if they satisfy the

user's request. This additional processing can be more general than the processing performed at the searchlet level (which was constrained to the independent evaluation of a single object). For instance, the additional processing may include cross-object correlations and auxiliary databases. Once the domain application determines that a particular object matches the user's criteria, the object is shown to the user. When processing a large data set, it is important to present the user with results as soon as they appear. Based on these partial results, the user can refine the query and restart the search. Query refinement leads to the generation of a new searchlet, which is once again executed by the back end.

3.1 Searchlets

3.1.1 Searchlet Structure

The searchlet is an application-specific proxy that Diamond uses to implement early discard. It consists of a set of *filters* and some configuration state (*e.g.*, filter parameters and dependencies between filters). For example, a searchlet to retrieve portraits of people in dark business suits might contain two filters: a color histogram filter that finds dark regions and an image processing filter that locates human faces.

For each object, the runtime invokes each of the filters in an efficient order, considering both filter cost and selectivity (see Section 5.2). The return value from each filter indicates whether the object should be discarded, in which case the searchlet evaluation is terminated. If an object passes all of the filters in the searchlet, it is sent to the domain application.

Before invoking the first filter, the runtime makes a temporary copy of the object. This copy exists only until the object is discarded or the search terminates, allowing the filters to share state and computation without compromising the stored object.

One filter can pass state to another filter by adding attributes (implemented as name-value pairs) to the temporary object being searched. The second filter recovers this state by reading these attributes. If the second filter requires attributes written by the first filter, then the configuration must specify that the second filter depends on the first filter. The runtime ensures that filters are evaluated in an order that satisfies their dependencies.

The filter functions are sent as object code to Diamond. This choice of object code instead of alternatives, such as domain-specific languages, was driven by several factors. First, many real-world applications (*e.g.*, drug discovery) may contain proprietary algorithms where requiring source code is not an option. Second, we want to encourage developers to leverage existing libraries and applications to simplify the development process. For instance, our image retrieval application (described in Section 4) relies heavily on the OpenCV [8] image processing library.

Executing application-provided object code raises serious security and safety implications that are not specifically addressed by the current implementation. Existing techniques, such as processes, virtual machines, or software fault isolation [31], could be incorporated into future implementations. Additionally, Diamond never allows searchlets to modify the persistent (on-disk) data.

3.1.2 Creating Searchlets

Searchlets can be generated by a domain application in response to a user's query in a number of ways. The most straightforward method is for domain experts to implement a library of filter functions that are made available to the application. The user specifies a query by selecting the desired filters and setting their parameters (typically using a GUI). The application determines filter dependencies and assembles the selected functions and parameters into a searchlet. This works well for domains where the space of useful early discard algorithms is well spanned by a small set of functions (potentially augmented by a rich parameter space). These functions could also be provided (in binary form) by third-parties.

Alternately, the domain application could generate code on-the-fly in response to the user's query. One could envision an application that allows the user to manually generate searchlet code. We believe that the best method for searchlet creation is highly domain-dependent, and the best way for a human to specify a search is an open research question.

3.2 Key Interfaces

The Diamond architecture defines three APIs to isolate components: the searchlet API, the filter API and associative DMA. These are briefly summarized below.

- The *searchlet API* provides the interface that applications use to interact with Diamond. This API provides calls to query device capabilities, scope a search to a specific collection of data, load searchlets, and retrieve objects that match the search.
- The *filter API* defines the interface used by the filter functions to interact with the storage run-time environment. This API provides functions to read and write object contents, as well as functions to read and write object attributes to share state among filters. Any changes only affect the temporary version of the object.
- *Associative DMA* isolates the host and the storage implementations. This API abstracts the transport mechanism and flow control between host and storage run-time systems. Associative DMA also provides a common interface that enables Diamond to employ diverse back-end implementations without modifications to the host runtime.

3.3 Host and Storage Systems

The host system is where the domain application executes. The user interacts with this application to formulate searches and to view results. Diamond attempts to balance computation between the host and storage systems. To facilitate this, storage devices may pass *unprocessed* objects to the host runtime, due to resource limitations or other constraints. The host runtime evaluates the searchlet, if necessary; if the object is not discarded, it is made available to the domain application. The storage system provides a generic infrastructure for searchlet execution; all of the domain-specific knowledge is completely encapsulated in the searchlet. This enables the same Diamond back-end to serve different domain applications (simultaneously, if necessary).

Diamond is well-suited for deployment on active storage, but such devices are not commercially available today, nor are they likely to become popular without compelling applications. Diamond provides a programming model that abstracts the storage system, enabling the development of applications that will seamlessly integrate with active storage devices as they become available.

Diamond's current design assumes that the storage system can be treated as object storage [30]. This allows the host to be independent of the data layout on the storage device and should allow us to leverage the emerging object storage industry standards.

4 Diamond Applications

Diamond provides a general framework for building interactive search applications. All of the domain-specific knowledge is contained in the front-end application and in the searchlets. To illustrate the process, consider the problem of drug candidate design.

Given a target protein, a chemist must search through a large database of 3D ligand structures to identify candidates that may associate strongly with the target. Since accurate calculation of the binding free energy of a particular ligand is prohibitively expensive, such programs could benefit from user input to guide the search in two ways. First, the chemist could adjust the granularity of the search (trading accuracy for speed). Second, the chemist could test hypotheses about a particular ligand-protein interaction using interactive molecular dynamics [14]. In Diamond, the former part of the search could be downloaded to the storage device while the latter could be performed on the chemist's graphical workstation. Early discard would reject hopeless ligands from consideration allowing the chemist to focus attention on the more promising candidates. If none of the initial candidates proved successful, the chemist would refine the search to be less selective. This example illustrates some of the characteristics that make an application suitable for early discard. First, that the user is searching for *specific instances* of data that match a query rather than aggregate statistics about the set of matching data items. Second, that the user's criteria for a successful match is often subjective, potentially ill-defined, and typically influenced by the partial results of the query. Third, that the mapping between the user's needs and the matching objects is too complex for it to be captured by a batch operation. An everyday example of such a domain is image search; the remainder of this section presents SnapFind, a prototype application for this domain built using the Diamond programming model.

4.1 SnapFind Description

SnapFind was motivated by the observation that digital cameras allow users to generate thousands of photos, yet few users have the patience to manually index them. Users typically wish to locate photos by semantic content (*e.g.*, "show me photos from our whale watching trip in Hawaii"); unfortunately, computer vision algorithms are currently incapable of understanding image content to this degree of sophistication. SnapFind's goal is to enable users to interactively search through large collections of unlabeled photographs by quickly specifying searchlets that roughly correspond to semantic content.

Research in image retrieval has attracted considerable attention in recent years [11,29]. However, prior work in this area has largely focused on whole-image searches. In these systems, images are typically processed off-line and compactly represented as a multi-dimensional vector. In other systems, images are indexed offline into several semantic categories. These enable users to perform interactive queries in a computationally-efficient manner; however, they do not permit queries about local regions *within* an image since indexing every subregion within an image would be prohibitively expensive. Thus, whole-image searches are well-suited to queries corresponding to general image content (*e.g.*, "find me an image of a sunset") but poor at recognizing objects that only occupy a portion of the image (*e.g.*, "find me images of people wearing jeans"). SnapFind exploits Diamond's ability to exhaustively process a data set using customized filters, enabling users to search for images that contain the desired content only in a small patch. The remainder of this section describes SnapFind and presents an informal validation of early discard.

SnapFind allows users to create complex image queries by combining simple filters that scan images for patches containing particular color distributions, shapes, or visual textures (detailed in a technical report [19]). The user can either select a pre-defined filter from a palette (*e.g.*, "frontal human faces" or "ocean waves") or create new filters by clicking on sample patches in other images (*e.g.*, creating a "blue jeans" color filter

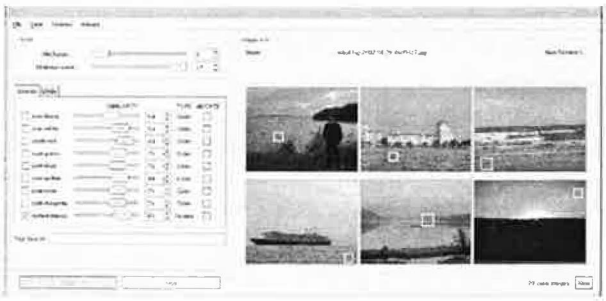


Figure 3: **SnapFind** – a proof-of-concept image search application designed using the Diamond programming model. Users can search a large image collection using customized filters. Here, the user has filtered for regions containing water texture (marked with white boxes). The filter correctly identifies most of the water in the images, but occasionally makes errors; for example, the sky in the bottom right image is incorrectly labeled as water.

by giving half a dozen examples). Indexing is infeasible for two reasons: (1) the user may define new search filters at query time; (2) the content of the patches is typically high-dimensional. When the user submits the query, SnapFind generates a searchlet and initiates a Diamond search. Diamond typically executes a portion of the query at the storage device, enabling early discard to reject many images in the initial stages of the pipeline.

4.2 SnapFind Usage Experience

We designed some simple experiments to investigate whether early discard can help exhaustive search. Our chosen task was to retrieve photos from an unlabeled collection based on semantic content. This is a realistic problem for owners of digital cameras and is also one that untrained users can perform manually (given sufficient patience). We explored two cases: (1) purely manual search, where all of the discard happens at the user stage; (2) using SnapFind. Both scenarios used the same graphical interface (see Figure 3), where the user could examine six thumbnails per page, magnify a particular image (if desired) and mark selected images.

Our data set contained 18,286 photographs (approximately 10,000 personal pictures, 1,000 photos from a corporate website, 5,000 images collected from an ethnographic survey and 2,000 from the Corel image CD-ROMs). These were randomly distributed over twelve emulated active storage devices. Users were given three minutes to tackle each of the following two queries: (S1) find images containing windsurfers or sailboats; (S2) find pictures of people wearing dark business suits or tuxedos.

For the manual scenario, we recorded the number of images selected by the user (correct hits matching the query) along with the number of images that the user viewed in the alloted time. Users could page through the images at their own pace, and Table 1 shows that users scanned the images rapidly, viewing 600–1,000 images in three minutes. Even at this rate of 2–5 images per second, they were only able to process about 5% of the total data.

	MANUAL		DIAMOND			
	hits	user seen	hits	user seen	system seen	early discard
S1						
a	7	684	46	396	18,286	97.8%
b	8	774	39	396	18,286	97.8%
c	13	966	46	396	18,286	97.8%
S2						
a	29	600	29	78	15,286	99.5%
b	18	612	29	74	15,362	99.5%
c	24	630	29	74	15,198	99.5%

Table 1: **SnapFind user study** - Results of an informal interactive search experiment using SnapFind. Users (a,b,c) were given three minutes to locate photographs matching each query (S1 and S2) in a collection of 18,286 images.

For the SnapFind scenario, the user constructed early discard searchlets simply by selecting one or more image processing filters from a palette of pre-defined filters, configured filter parameters using the GUI, and combined them using boolean operators. Images that satisfied the filtering criteria (*i.e.*, those matching a particular color, visual texture or shape distribution in a subregion) were shown to the user. Based on partial results, the user could generate a new searchlet by selecting different filters or adjusting parameters. As in the manual scenario, the user then marked those images that matched the query. For S1, the early discard searchlet was a single "water texture" filter trained on eight 32×32 patches containing water. For S2, the searchlet was a conjunction of a color histogram filter combined with a face detector. Table 1 shows these results, and searchlets are detailed in Table 2.

For S1, SnapFind significantly increases the number of relevant images found by the user. Diamond is able to exhaustively search through all of the data, and early discard eliminates almost 98% of the objects at the storage devices. This shows how early discard can help users find a greater number of relevant objects.

For S2, the improvement, as measured by hits alone, is less dramatic, but early discard shows a different benefit. Although Diamond fails to complete the exhaustive search in three minutes (it processes about 85% of the data), the user achieves approximately as many hits as in the manual scenario while viewing 88% fewer images. For applications where the user only needs a few images, early discard is ideal because it significantly decreases the user's effort. By displaying fewer irrelevant

items, the user can devote more attention to the promising images.

5 Prototype Implementation

Our Diamond prototype is currently implemented as user processes running on Red Hat Linux 9.0. The searchlet API and the host runtime are implemented as a library that is linked against the domain application. The host runtime and network communication are threads within this library. We emulate active storage devices using off-the-shelf server hardware with locally-attached disks. The active storage system is implemented as a daemon. When a new search is started, new threads are created for the storage runtime and to handle network and disk I/O. Diamond's object store is implemented as a library that stores objects as files in the native file system. Associative DMA is currently under definition; Diamond uses a wrapper library built on TCP/IP with customized marshalling functions to minimize data copies.

The remainder of this section details Diamond's two primary mechanisms for efficient early discard: run-time partitioning of computation between the host and storage devices, and dynamic ordering of filter evaluation to reject undesirable data items as efficiently as possible.

5.1 Dynamic Partitioning of Computation

As discussed in Section 2, bottlenecks in exhaustive search pipelines are not static. Diamond achieves significant performance benefits by dynamically balancing the computational task between the active storage devices and the host processor.

The Diamond storage runtime decides whether to evaluate a searchlet locally or at the host computer. This decision can be different for each object, allowing the system to have fine-grained control over the partitioning. Thus, even for searchlets that consist of a single monolithic filter, Diamond can partition the computation on a per-object basis to achieve the ratio of processing between the storage devices and the host that provides the best performance. The ability to make these fine-grained decisions is enabled by Diamond's assumption that objects can be processed in any order, and that filters are stateless.

If the searchlet consists of multiple filters, Diamond could partition the work so that some filters execute on the storage devices and the remainder execute on the host; the current implementation does not consider such partitionings. Diamond could also detect when there are many objects waiting for user attention and choose to evaluate additional filters to discard more objects.

The current implementation supports two methods for partitioning computation between the host and the storage devices. The effectiveness of these methods in practice is evaluated in Section 6.3.

5.1.1 CPU Splitting

In this method, the host periodically estimates its available compute resources (processor cycles), determines how to allocate them among the storage devices, and sends a message to each device. The storage device receives this message, estimates its own computational resources, and determines the percentage of objects to process locally. For example, if a storage device estimates that it has 100 MIPS and receives a slice of 50 MIPS from the host, then it should choose to process 2/3 of the objects locally and send the remaining (unprocessed) objects to the host. CPU splitting has a straightforward implementation: whenever the storage runtime reads an object, it probabilistically decides whether to process the object locally.

5.1.2 Queue Back-Pressure

Queue Back-Pressure (QBP) exploits the observation that the length of queues between components in the search pipeline (see Figure 1) provide valuable information about overloaded resources. The algorithm is implemented as follows.

When objects are sent to the host, they are placed into a work queue that is serviced by the host runtime. If the queue length exceeds a particular threshold, the host refuses to accept new objects. Whenever the storage runtime has an object to process, it examines the length of its transmit queue. If the queue length is less than a threshold, the object is sent to the host without processing. If the queue length is above the threshold, the storage runtime evaluates the searchlet on the object. This algorithm dynamically adapts the computation performed at the storage devices based on the current availability of downstream resources. When the host processor or network is a bottleneck, the storage device performs additional processing on the data, easing the pressure on downstream resources until data resumes its flow. Conversely, if the downstream queues begin to empty, the storage runtime will aggressively push data into the pipeline to prevent the host from becoming idle.

5.2 Filter ordering

A Diamond searchlet consists of a set of filters, each of which can choose to discard a given object. We assume that the set of objects that pass through a particular searchlet is completely determined by the set of filters in the searchlet (and their parameters). However, the filter order dramatically impacts the efficiency with which Diamond processes a large amount of data.

Diamond attempts to reorder the filters within a searchlet to run the most promising ones early. Note that the best filter ordering depends on the set of filters, the user's query, and the particular data set. For example, consider a user who is searching a large image collection for photos of people in dark suits. The application may

determine that a suitable searchlet for this tasks includes two filters (see Table 3): a face detector that matches images containing human faces (filter F1); and a color filter that matches dark regions in the image (filter F3). From the table, it is clear that F1 is more selective than F3, but also much more computationally expensive. Running F1 first would work well if the data set contained a large number of night-time photos (which would successfully pass F3). On the other hand, if the collection contained a large number of baby pictures, running F1 early would be extremely inefficient.

The effectiveness of a filter depends upon its selectivity (pass rate) and its resource requirements. The total cost of evaluating filters over an object can be expressed analytically as follows. Given a filter, Fi, let us denote the cost of evaluating the filter as $c(Fi)$, and its pass rate as $\mathrm{P}(Fi)$. In general, the pass rates for the different filters may be correlated (*e.g.*, if an image contains a patch with water texture, then it is also more likely to pass through a blue color filter). We denote the *conditional pass rate* for a filter Fi that is processing objects that successfully passed filters Fa, Fb, Fc by $\mathrm{P}(Fi|Fa,Fb,Fc)$. The average time to process an object through a series of filters $F0 \ldots Fn$ is given by the following formula:

$$C = c(F0) + \mathrm{P}(F0)c(F1) + \mathrm{P}(F1|F0)\mathrm{P}(F0)c(F2) + \mathrm{P}(F2|F1,F0)\mathrm{P}(F1|F0)\mathrm{P}(F0)c(F3) + \cdots$$

The primary goal of choosing a filter order is to minimize this cost function. To perform this optimization, the system needs the costs of the different filters and the conditional pass rates. Diamond estimates these values during a searchlet execution by varying the order the filters are evaluated and measuring the pass rates and costs over a number of objects.

5.2.1 Partial Orderings

Allowing filters to use results generated by other filters enables searchlets to: (1) use generic components to compute well-known properties; (2) reuse the results of other filters. For instance, all of the color filters in SnapFind (see Section 4) rely on a common data structure that is generated by an auxiliary filter. Filter developers can explicitly specify the attributes that each filter requires, and these dependencies are expressed as *partial ordering* constraints. Figure 4 shows an example of a partial order. The forward arrows indicate an 'allows' relationship. For example, "Reader" is a prerequisite for "Histogram" and "WaterTexture", and "Red" and "Black" are prerequisites for "ColorTest". The filter ordering problem is to find a *linear extension* of the partial order. Figure 5 shows one possible order. Note that finding a path through this directed acyclic graph is not sufficient; all of the filters in the searchlet still need to be evaluated.

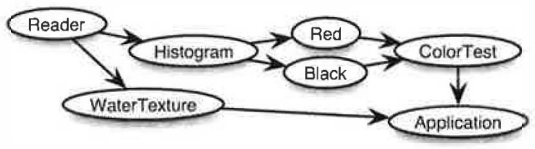


Figure 4: **Example partial ordering** - "Reader" must be executed before "Histogram" and "WaterTexture". "Histogram" must be evaluated before "Red" and "Black".

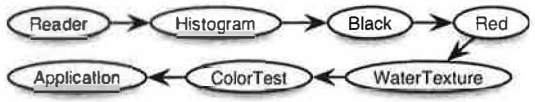


Figure 5: **Linear Extension** - a possible ordering for the filters shown in Figure 4.

5.2.2 Ordering Policies

The *filter ordering policy* is the method that Diamond uses for choosing the sequence for evaluating the filters. We describe three policies below.

- **Independent:** assumes that there is no correlation between $\mathrm{P}(Fi)$ and $\mathrm{P}(Fj)$, or $c(Fi)$ and $c(Fj)$. Using this assumption we can find an optimal sequence by sorting on $c(Fi)/\mathrm{P}(Fi)$ [24]. In practice the correlations between filters may cause this policy to perform poorly.
- **Hill climbing (HC):** picks a random sequence from the space of all legal linear extensions. The policy attempts to iteratively improve the order by swapping adjacent filters until a local minimum is reached. Multiple random restarts are used to reduce sensitivity to the starting point.
- **Best filter first (BFF):** iteratively expands a list of valid sub-sequences to find the optimal filter sequence. BFF initializes a list with the set of single-element sub-sequences consisting of the filters that have no dependencies. BFF then removes the cheapest sub-sequence from the list, computes all valid sub-sequences that are one filter longer, and reinserts them into the list. BFF terminates when the cheapest sub-sequence is complete; this is an optimal sequence. The algorithm is motivated by the observation that later filters typically have less impact on the average cost than earlier filters, because the overall pass probability drops as one goes deeper in the filter chain.

6 Experimental Evaluation

This section presents empirical results from a variety of experiments using SnapFind running on the Diamond implementation described in Section 5. The active storage devices were emulated using rack-mounted computers (1.2 GHz Intel® Pentium® III processors, 512 MB RAM and 73 GB SCSI disks), connected via a 1 Gbps Ethernet switch. The host system contained a 3.06 GHz Intel® Pentium® Xeon™ processor, 2 GB RAM, and a 120 GB IDE disk. The host was connected via Ethernet to the storage platforms. We varied the link speed between 1 Gbps and 100 Mbps depending on the experiment. Some experiments required us to emulate slower active storage devices; this was done by running a real-time task that consumed a fixed percentage of the CPU. These experiments employ homogeneous backends.

6.1 Description of Searchlets

We evaluate Diamond using the set of queries enumerated in Table 2. These consist of real queries from SnapFind searches supplemented by several synthetic queries. The searchlets are described in Table 2, and the filters used by these searchlets are listed in Table 3.

The Water (S1) and Business Suits (S2) queries match the tasks we used in Section 4. The Halloween (S3) query is similar to Business Suits with an additional filter. The three synthetic queries (S4–S6) are used to test filter ordering and the two Dark Patch queries (S7, S8) are used to illustrate bottlenecks for dynamic partitioning.

Table 3 provides a set of measurements summarizing the discard rate and the computational cost of running the various filters. We determined these filter characteristics by evaluating each filter over the objects in our image collection (described in Section 4). The overall discard rate is the fraction of objects dropped divided by the total number of images, and the cost is the average number of CPU milliseconds consumed. Filters F0–F5 are taken from SnapFind. The other filters were synthetically generated with specific characteristics.

The searchlets S5 and S6 were designed to examine the effect of filter correlation. F14, F15 and F16 are correlated: $P(F14,F15,F16) \neq P(F14)P(F15)P(F16)$. F17, F18 and F19 are uncorrelated: $P(F17,F18,F19) = P(F17)P(F18)P(F19)$.

6.2 Disk and Host Processing Power

Our first measurements examine how variations in system characteristics (number of storage devices, interconnect bandwidth, processor performance, queries) affect the average time needed to process each object. For each configuration, we measure the completion time for a different static partitioning between the host and storage devices. A particular partition is identified by percentage of objects that are evaluated at the storage devices. Remaining objects are passed to the host for processing.

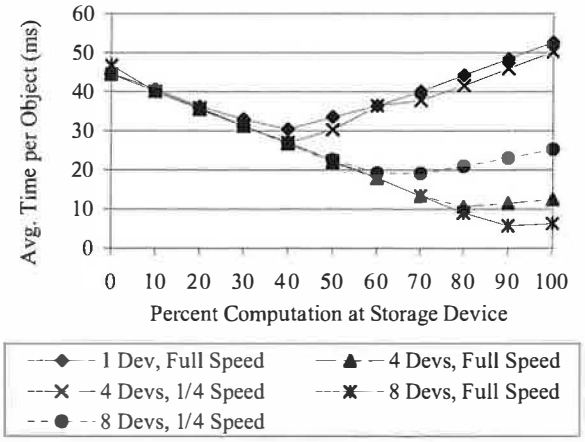


Figure 6: **Compute Limited** - This graph shows how the average time spent processing an object varies with the percentage of the objects evaluated at the storage system when the CPU is the bottleneck. The average time is computed as the total search time divided by the number of objects searched.

In these experiments, each storage device has 5,000 objects (1.6 GB). As the number of storage devices increases, so does the total number of objects involved in a search. For each configuration, we report the mean time needed for Diamond to process each object (averaged over three runs). Our data set was chosen to be large enough to avoid startup transients but small enough to enable many different experiments. Using a larger data set would give the same average time per object, but will increase the overall completion time for a search.

The first set of experiments (see Figure 6) shows how variations in the relative processing power of the host and storage devices affect search time for CPU-bound tasks. These experiments use searchlet S3 to find pictures of a child in a Halloween costume.

We observe that, as the number of storage devices increases, more computation is moved to the storage devices. This matches our intuition that as the aggregate processing power of the storage devices increases, more of the overall processing should be done at the storage devices.

When there is no processing at the storage devices, this is equivalent to reading all of the data from network storage. On the left-hand side of the lines, we see linear decreases as processing is moved to the storage devices, reducing the load on the bottleneck. When most of the processing moves to the storage device, the bottleneck becomes the storage device and we see increases in average processing time. The best case is the local minimum; this corresponds to the case where the load be-

Query		Searchlet Description	CPU Cost
Water - regions containing water waves	S1	Uses texture filter trained on water samples.	Low
Business Suits - images of people in dark business suits	S2	Uses face detector and color histogram trained on dark patches of color.	High
Halloween - images of a child in Halloween costume	S3	Uses face detector and color histograms trained on red patches and dark patches of color.	High
Synthetic	S4	Synthetic filters with inversely (non-linearly) related pass rate and cost.	Med
Synthetic	S5	Three filters with correlated pass rate and constant cost.	Low
Synthetic	S6	Three filters with independent pass rate (same as S5 overall) and constant cost.	Low
Dark Patch A - searchlet with high selectivity	S7	Uses color histogram trained on black sample patch; has a high threshold so few images match.	Low
Dark Patch B - searchlet with low selectivity	S8	Uses color histogram trained on black sample patch; has a low threshold so many images match.	Low

Table 2: **Test Queries** - The queries and associated searchlets used to evaluate the Diamond prototype.

Filter	Searchlet	Discard rate	CPU (ms)
F0 - Reader (required)	S1,2,3,7,8	0	5
F1 - Face Detect	S2,3	99%	530
F2 - Histogram	S2,3,7,8	0	20
F3 - Black (req. F2)	S2	83%	2
F3a - Black (req. F2)	S7	99%	2
F3b - Black (req. F2)	S8	78%	2
F4 - Red (req. F2)	S2	99%	2
F5 - Wave Texture	S1	95%	14
F6 - Synthetic		20%	2
F7 - Synthetic		22%	4
F8 - Synthetic		26%	8
F9 - Synthetic	S4	29%	16
F10 - Synthetic		31%	32
F11 - Synthetic		33%	64
F12 - Synthetic		36%	128
F13 - Synthetic		36%	256
F14		50%	1
F15 - Synthetic	S5	40%	8
F16		30%	8
F17		50%	1
F18 - Synthetic	S6	40%	8
F19		30%	8

Table 3: **Filters** - The discard rate is over a collection of 18,286 images. F0 and F2 are helper filters that do not discard data. The filters in S5 are correlated; those in S6 are uncorrelated.

tween the host and the storage devices is balanced. Note that Diamond benefits from active storage even with a small number of storage devices.

Our next measurements (see Figure 7) examine the network-bound case using searchlets S7 and S8. Both searchlets look for a small dark region and are relatively cheap to compute. S7 rejects most of the objects (highly selective) while S8 passes a larger fraction of the objects (non-selective).

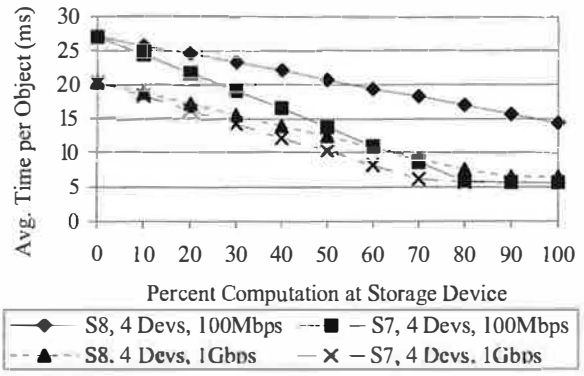


Figure 7: **Network Limited** - This graph shows how the average time per object varies with the percentage of objects evaluated at the storage system when the network is the bottleneck. The average time is computed as the total search time divided by the number of objects searched.

These experiments demonstrate that as the network becomes the limiting factor, more computation should be performed at the storage device. We also see that these lines flatten out at the point where reading the data from the disk becomes a bottleneck. The upper two lines show S7 and S8 running on a 100 Mbps network. We see that S8 is always slower, even when all of the computation is performed at the storage device. This is because S8 passes a large percentage of the objects, creating a data transfer bottleneck in all cases.

6.3 Impact of Dynamic Partitioning

This section evaluates the effectiveness of the dynamic partitioning algorithms presented in Section 5. As a baseline measurement, we manually find the ideal partitioning based on the results from the previous section. We then compare the time needed to complete

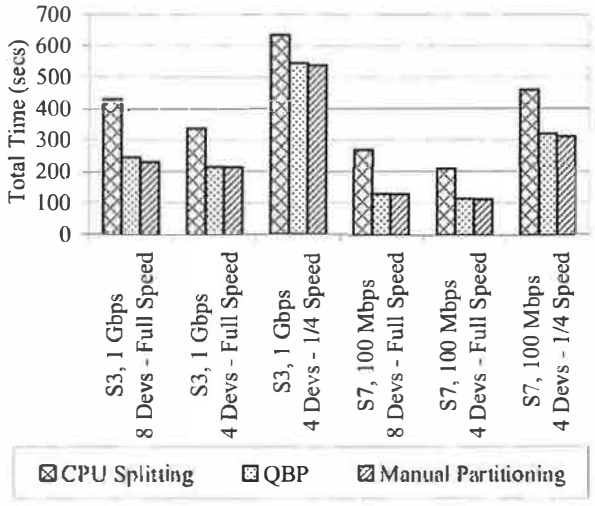


Figure 8: **Dynamic Partitioning** - This graph compares the performance of two automated partitioning algorithms against a hand-tuned manual partitioning.

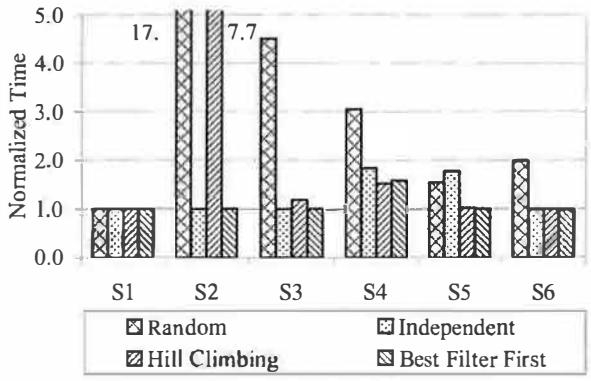


Figure 9: **Filter Ordering** - Execution time for evaluating searchlets using different ordering policies, normalized to the Offline Best policy.

the search using this manual partitioning to those for the two dynamically-adjusting schemes: *CPU Splitting* and *Queue Back-Pressure* (QBP).

For these tests, we use both a CPU-bound task (searchlet S3) and a network-bound task (searchlet S7). We run each task in a variety of configurations and compare the results as shown in Figure 8.

In all of these cases, the QBP technique gives similar performance to the Best Manual technique. CPU Splitting does not perform as well in most of the cases for two reasons. First, in the network-bound task (searchlet S7), the best results are obtained by processing all objects at the storage devices. CPU Splitting always tries to process a fraction of the objects on the host, even when sending data to the host is the bottleneck. QBP detects the network bottleneck and processes the objects locally. Second, relative CPU speeds are a poor estimate of the time needed to evaluate the filters. Most of these searchlets involve striding over large data structures (images) so the computation tends to be bound by memory access, not CPU. As a result, increasing the CPU clock rate does not give a proportional decrease in time. It is possible that more sophisticated modeling would make CPU Splitting more effective. However, given that the simple QBP technique works so well, there is probably little benefit to pursuing that idea.

6.4 Impact of Filter Ordering

This section compares the different policies described in Section 5.2.2, and illustrates the significance of filter ordering. We use searchlets S1–S6, which are composed of the filters detailed in Table 3. This experiment eliminates network and host effects by executing entirely on a single storage device and compares different local optimizations. Total time is normalized to the *Offline Best* policy; this is the best possible static ordering (computed using an oracle), and provides a bound on the minimum time needed to process a particular searchlet. *Random* picks a random legal linear order at regular intervals. This is the simplest solution that avoids adversarial worst cases without extra state, and would be a good solution if filter ordering did not matter.

Figure 9 shows that completion time varies significantly with different filter ordering policies. The poor performance of *Random* demonstrates that filter ordering is significant. There is a unique legal order for S1, and all methods pick it correctly. *Independent* finds the optimal ordering when filters are independent, as in S6, but can generate expensive orderings when they are not, as in S5. *Hill Climbing* sometimes performs poorly because it can get trapped in local minima. *Best Filter First* is a dynamic techniques that works as well as *Independent* (it has a slightly longer convergence time) with independent filters, and has good performance with dependent filters. The dynamic techniques spend time exploring the search space, so they always pay a penalty over the *Offline Best* policy. This is more pronounced with more filters, as in S4.

The next experiment examines Diamond performance when dynamic partitioning and filter ordering are run concurrently. For our baseline measurement, we manually find the best partitioning and the best filter ordering for each configuration. We then compare the time needed to execute searchlet S3 against two test cases that use dynamic adaptation. The first uses dynamic partitioning (QBP) and the filter ordering (BFF); the second uses dynamic partitioning (QBP) and randomized filter orders. Figure 10 shows the results of these experiments. As expected, the combination of dynamic partitioning and dynamic filter ordering gives us results that are close to the best manual partitioning. Random filter ordering performs less well because of the longer time needed to

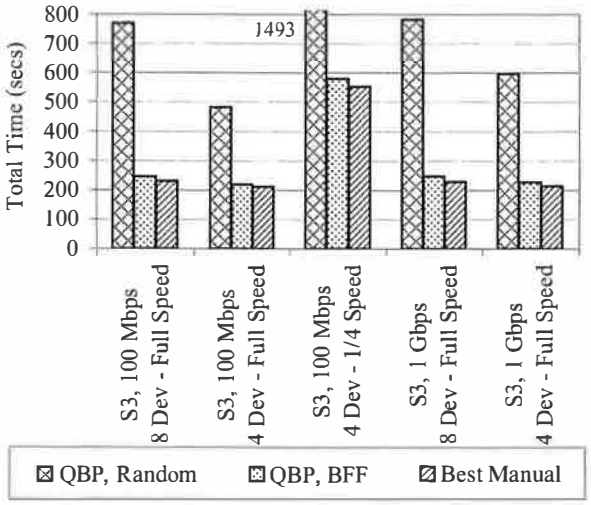


Figure 10: **Dynamic Optimizations** - Execution times for evaluating searchlets using a combination of dynamic partitioning and filter ordering, compared against a hand-tuned algorithm.

process each object.

6.5 Using Diamond on Large Datasets

To better understand the impact of Diamond on real-world problems, we consider a typical scenario: how much data could a user search in an afternoon? The results from Figure 10 show that Diamond can process 40,000 objects (8 storage devices with 5,000 objects each) in 247 seconds. Thus, given four hours, the user should be able to search through 2.3 million objects (approximately 0.75 TB) using the same searchlets. In the case of searchlet S3, this would imply that the user should see about 115 objects. However, since the number of objects seen by the user is sensitive to search parameters and the distribution of data on the storage device, it could vary greatly from this estimate.

Although raw performance should scale as disks are added, the limitations imposed by the user and the domain application are less clear. For instance, in the drug discovery application described in Section 4, the user's think-time may be the limiting factor even when Diamond discards most of the data. Conversely, in other domains, the average computational cost of a searchlet could be so high that Diamond would be unable to process all of the data in the alloted time. These questions are highly domain-dependent and lie beyond the scope of this paper.

As we discussed in the introduction, the current implementation is focused on pure brute-force search, but other complimentary techniques can be used to improve performance. One technique would be to use pre-computed indices to reduce the number of objects searched. For example, filter F1 from Table 3 could be used to build an index of pictures containing faces. Using this index would reduce the search space by 99% for any searchlets that use filter F2.

Another complimentary technique is to take advantage of cached results. In certain domains, a user may frequently refine a searchlet based on partial results in a manner that leaves most of its filters and their parameters unchanged. For instance, in SnapFind, the user may modify a search by adding a filter to the existing set of filters in the searchlet. When re-executing a filter with the same parameters, Diamond could gain significant computational benefits by retrieving cached results. However, caching may provide very little benefit for other applications. For instance, a Diamond application that employs relevance feedback [16] typically adjusts filter arguments at each iteration based on user-provided feedback. Since the filter arguments are different with each search, the use of cached information becomes more difficult. We plan to evaluate the benefits of caching as we gain more experience with other Diamond applications.

7 Related Work

To the best of our knowledge, Diamond is the first attempt to build a system that enables efficient interactive search of large volumes of complex, non-indexed data. While unique in this regard, Diamond does build upon many insights and results from previous work.

Recent work on *interactive data analysis* [15] outlines a number of new technologies that will be required to make database systems as interactive as spreadsheets — requiring advances in databases, data mining and human-computer interaction. Diamond and early discard are complementary to these approaches, providing a basic systems primitive that furthers the promise of interactive data analysis.

In more traditional database research, advanced indexing techniques exist for a wide variety of specific data types including multimedia data [10]. Work on data cubes [13] takes advantage of the fact that many decision support queries are well-known to pre-process a database and then perform queries directly from the more compact representation. The developers of new indexing technology must constantly keep up with new data types, and with new user access and query patterns. A thorough survey of indexing and the outline of this tension appear in a recent dissertation [27], which also details theoretical and practical bounds on the (often high) cost of indexing.

Work on *approximate query processing*, recently surveyed in [5], complements these efforts by observing that users can often be satisfied with approximate answers when they are simply using query results to iter-

ate through a search problem, exactly as we motive in our interactive search tasks.

In addition, in high-dimensionality data (such as feature vectors extracted from images to support indexing), sequential scanning is often competitive with even the most advanced indexing methods because of the *curse of dimensionality* [6, 9, 33]. Efficient algorithms for *approximate* nearest neighbor in certain high-dimensional spaces, such as locality-sensitive hashing [12], are available. However, these require the similarity metric be known in advance (so that the data can be appropriately pre-indexed using the proximity-preserving hashing functions) and that the similarity metric satisfy certain properties. Diamond addresses searches where neither of these constraints is satisfied.

In systems research, our work builds on the insight of active disks [1, 20, 25] where the movement of search primitives to extended-function storage devices was analyzed in some detail, including for image processing applications. Additional research has explored methods to improve application performance using active storage [21, 22, 26, 32]. The work of Abacus [2], Coign [18], River [3] and Eddies [4] provide a more dynamic view in heterogeneous systems with multiple applications or components operating at the same time. Coign focuses on communication links between application components. Abacus automatically moves computation between hosts or storage devices in a cluster based on performance and system load. River handles adaptive dataflow control generically in the presence of failures and heterogeneous hardware resources. Eddies [4] adaptively reshapes dataflow graphs to maximize performance by monitoring the rates at which data is produced and consumed at nodes. The importance of filter ordering has also been the object of research in database query optimization [28]. The addition of early discard and filter ordering bring a new set of semantic optimizations to all of these systems, while retaining the basic model of observation and adaptation while queries are running.

Recent efforts to standardize object-based storage devices (OSD) [30] provide the basic primitives on which we build our semantic filter processing. In order to most efficiently process searchlets, active storage devices must contain whole objects, and must understand the low-level storage layout. We can also make use of the attributes that can be associated with objects to store intermediate filter state and to save filter results for possible re-use in future queries. Offloading space management to storage devices provides the basis for understanding data in the more sophisticated ways necessary for early discard filters to operate.

8 Conclusion

Diamond is a system that supports interactive data analysis of large complex data sets. This paper argues that these applications require applying brute-force search to a portion of the objects. To efficiently perform brute-force search the Diamond architecture uses *early discard* to push filter processing to the edges of the system — executing semantic data filters directly at storage devices, and greatly reducing the flow of data into the central bottlenecks of a system. The Diamond architecture also enables the system to adapt to different hardware configurations by dynamically adjusting where computation is performed.

To validate our architecture, we have implemented a prototype version of Diamond and an application, SnapFind, that interactively searches collections of digital images. Using this system, we have demonstrated that searching large collections of images is feasible and that the system can dynamically adapt to use the available resources such as network and host processor efficiently.

In the future, we plan to work with domain experts to create new interactive search applications such as ligand screening or satellite imagery analysis. Using these applications, we plan to validate our approach to interactive search of large real-world datasets.

Acknowledgments

Thanks to: Derek Hoiem and Padmanabham (Babu) Pillai for their valuable help with the Diamond system; Genevieve Bell and David Westfall for contributing data for the SnapFind user study; Ben Janesko and David Yaron for useful discussions on applying Diamond to computational chemistry problems; our shepherd, Christos Karamanolis for all his help; and the anonymous reviewers for feedback on an earlier draft of the paper.

References

[1] ACHARYA, A., UYSAL, M., AND SALTZ, J. Active disks: Programming model, algorithms and evaluation. In *Proceedings of ASPLOS* (1998).

[2] AMIRI, K., PETROU, D., GANGER, G., AND GIBSON, G. Dynamic function placement for data-intensive cluster computing. In *Proceedings of USENIX* (2000).

[3] ARPACI-DUSSEAU, R., ANDERSON, E., TREUHAFT, N., CULLER, D., HELLERSTEIN, J., PATTERSON, D., AND YELICK, K. Cluster I/O with River: Making the fast case common. In *Proceedings of Input/Output for Parallel and Distributed Systems* (1999).

[4] AVNUR, R., AND HELLERSTEIN, J. Eddies: Continuously adaptive query processing. In *Proceedings of SIGMOD* (2000).

[5] BABCOCK, B., CHAUDHURI, S., AND DAS, G. Dynamic sample selection for approximate query processing. In *Proceedings of of SIGMOD* (2003).

[6] BERCHTOLD, S., BOEHM, C., KEIM, D., AND KRIEGEL, H. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of PODS* (May 1997).

[7] BORAL, H., AND DEWITT, D. Database machines: an idea whose time has passed? A critique of the future of database machines. In *Proceedings of of the International Workshop on Database Machines* (1983).

[8] BRADSKI, G. Programmer's tool chest: The OpenCV library. *Dr. Dobbs Journal* (November 2000).

[9] DUDA, R., HART, P., AND STORK, D. *Pattern Classification.* Wiley, 2001.

[10] FALOUTSOS, C. *Searching Multimedia Databases by Content.* Kluwer Academic Inc., 1996.

[11] FLICKNER, M., SAWHNEY, H., NIBLACK, W., ASHLEY, J., HUANG, Q., DOM, B., GORKANI, M., HAFNER, J., LEE, D., PETKOVIC, D., STEELE, D., AND YANKER, P. Query by image and video content: the QBIC system. *IEEE Computer 28* (1995).

[12] GIONIS, A., INDYK, P., AND MOTWANI, R. Similarity search in high dimensions via hashing. In *Proceedings of VLDB* (1999).

[13] GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., AND VENKATRAO, M. Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery 1* (1997).

[14] GRAYSON, P., TAJKHORSHID, E., AND SCHULTEN, K. Mechanisms of selectivity in channels and enzymes studied with interactive molecular dynamics. *Biophysical Journal 85* (2003).

[15] HELLERSTEIN, J., AVNUR, R., CHOU, A., HIDBER, C., RAMAN, V., ROTH, T., AND HAAS, P. Interactive data analysis: The CONTROL project. *IEEE Computer* (August 1999).

[16] HOIEM, D., SUKTHANKAR, R., SCHNEIDERMAN, H., AND HUSTON, L. Object-based image retrieval using the statistics of images. Tech. Rep. Intel Research IRP-TR-03-13, November 2003.

[17] HSIAO, D. Database machines are coming, database machines are coming. *IEEE Computer 12*, 3 (1979).

[18] HUNT, G., AND SCOTT, M. The Coign automatic distributed partitioning system. In *Proceedings of OSDI* (1999).

[19] HUSTON, L., SUKTHANKAR, R., R. WICKREMESINGHE, M. S., GANGER, G., RIEDEL, E., AND AILAMAKI, A. Diamond: A storage architecture for early discard in interactive search. Tech. Rep. IRP-TR-2004-02, Intel Research, January 2004. `http://www.intel-research.net/pittsburgh/publications.asp`.

[20] KEETON, K., PATTERSON, D., AND HELLERSTEIN, J. A case for intelligent disks (IDISKs). *SIGMOD Record 27*, 3 (1998).

[21] MA, X., AND REDDY, A. MVSS: An Active Storage Architecture. *IEEE Transactions On Parallel and Distributed Systems 14*, 10 (2003).

[22] MEMIK, G., KANDEMIR, M., AND CHOUDHARY, A. Design and evaluation of smart disk architecture for DSS commercial workloads. In *International Conference on Parallel Processing* (2000).

[23] MINKA, T., AND PICARD, R. Interactive learning using a society of models. *Pattern Recognition 30* (1997).

[24] PRUESSE, G., AND RUSKEY, F. Generating linear extensions fast. *SIAM Journal on Computing 23*, 2 (April 1994).

[25] RIEDEL, E., GIBSON, G., AND FALOUTSOS, C. Active storage for large-scale data mining and multimedia. In *Proceedings of VLDB* (August 1998).

[26] RUBIO, J., VALLURI, M., AND JOHN, L. Improving transaction processing using a hierarchical computing server. Tech. Rep. TR-020719-01, Laboratory for Computer Architecture, The University of Texas at Austin, July 2002.

[27] SAMOLADAS, V. *On Indexing Large Databases for Advanced Data Models.* PhD thesis, University of Texas at Austin, August 2001.

[28] SELINGER, P., ASTRAHAN, M., CHAMBERLIN, D., LORIE, R., AND PRICE, T. Access path selection in a relational database management system. In *Proceedings of SIGMOD* (1979).

[29] SMEULDERS, A., AND WORRING, M. Content-based image retrieval at the end of the early years. *IEEE Transactions on Pattern Analysis and Machine Intelligence 22*, 12 (2000).

[30] ANSI T10/1355-D: SCSI Object-Based Storage device commands (OSD), September 2003. `http://www.t10.org/ftp/t10/drafts/osd/`.

[31] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles* (December 1993).

[32] WICKREMISINGHE, R., VITTER, J., AND CHASE, J. Distributed computing with load-managed active storage. In *In IEEE International Symposium on High Performance Distributed Computing (HPDC-11)* (2002).

[33] YAO, A., AND YAO, F. A general approach to D-Dimensional geometric queries. In *Proceedings of STOC* (May 1985).

MEMS-based storage devices and standard disk interfaces: A square peg in a round hole?

Steven W. Schlosser, Gregory R. Ganger
Carnegie Mellon University

Abstract

MEMS-based storage devices (MEMStores) are significantly different from both disk drives and semiconductor memories. The differences motivate the question of whether they need new abstractions to be utilized by systems, or if existing abstractions will be sufficient. This paper addresses this question by examining the fundamental reasons that the abstraction works for existing devices, and by showing that these reasons also hold for MEMStores. This result is shown to hold through several case studies of proposed roles MEMStores may take in future systems and potential policies that may be used to tailor systems' access to MEMStores. With one noted exception, today's storage interfaces and abstractions are as suitable for MEMStores as for disks.

1 Introduction

MEMS-based storage devices (MEMStores) offer an interesting new component for storage system designers. With tiny mechanical positioning components, MEMStores offer disk-like densities (which are consistently greater than FLASH or MRAM projections) with order of magnitude latency and power reductions relative to high-performance and low-power disks, respectively. These features make MEMStores worthy of exploration now, so that designers are ready when the devices become available.

A debate has arisen, during this exploration, about which form of algorithms and interfaces are appropriate for MEMStores. Early work [11, 28] mapped the linear logical block number (*LBN*) abstraction of standard storage interfaces (SCSI and IDE/ATA) onto MEMStores, and concluded that MEMStores looked much like disks. From anecdotal evidence, it is clear that many researchers are unhappy with this; since MEMStore mechanics are so different from disks, they assume that MEMStores **must** need a new abstraction. Several groups [14, 29, 35] are exploring more device-specific approaches. As is often the case with such debates, we believe that each "side" is right in some ways and wrong in others. There is clearly a need for careful, balanced development of input for this debate.

We divide the aspects of MEMStore use in systems into two categories: roles and policies. MEMStores can take on various *roles* in a system, such as disk replacement, cache for hot blocks, metadata-only storage, etc. For the debate at hand, the associated sub-question is whether a system using a MEMStore is exploiting something MEMStore-specific (e.g., because of a particularly well-matched access pattern) or just benefitting from its general properties (e.g., that they are faster than current disks). In any given role, external software uses various *policies*, such as data layout and request scheduling, for managing underlying storage. The sub-question here is whether MEMStore-specific policies are needed, or are those used for disk systems sufficient.

The contribution of this paper is to address this core question about the use of MEMStores in systems:

> *Do MEMStores have unique, device-specific characteristics that a computer system designer should care about, or can they just be viewed as small, low-power, fast disk drives?*

Of course, MEMStores may realize performance and power characteristics that are unachievable with real disk technologies. The question restated, then, is: would a hypothetical disk, scaled from existing technology to the same average performance as a MEMStore, look the same to the rest of the system as a MEMStore? If MEMStores have characteristics that are sufficiently different from disk drives, then systems should use a different abstraction to customize their accesses to take advantage of the differences. If MEMStores do not have sufficiently different characteristics, then systems can simply treat MEMStores as fast disks and use the same abstraction for both.

To help answer this question, we use two simple objective tests. The first test, called the *specificity test*, asks: Is the potential role or policy truly MEMStore-specific? To test this, we evaluate the potential role or policy for both a MEMStore and a (hypothetical) disk drive of equivalent performance. If the benefit is the same, then the potential role or policy (however effective) is not truly MEMStore-specific and could be just as beneficial to disk drives. The second test, called the *merit test*, asks: Given that a potential role or policy passes the specificity test, does it make enough of an impact in performance (e.g., access speed or energy consumption) to justify a new abstraction? The test here is

a simple improvement comparison, e.g., if the system is less than 10% faster when using the new abstraction, then it's not worth the cost.

In most aspects, we find that viewing MEMStores as fast disks works well. Although faster than disks, MEMStores share many of their access characteristics. Signal processing and media access mechanisms strongly push for a multi-word storage unit, such as the ubiquitous 512 byte block used in disks. MEMStore seek times are strongly distance-dependent, correlated with a single dimension, and a dominant fraction of access time, motivating data layouts and scheduling algorithms that are similar to those used for disks. After positioning, sequential access is most efficient, just like in disks. The result is that most disk-based policies will work appropriately, without specialization for MEMStores, and that most roles could equally well be filled by hypothetical disks with equivalent average-case performance and power characteristics.

Our model of MEMStores is based on lengthy discussions with engineers who are designing the devices, and on an extensive study of the available literature. However, as MEMStores are not yet available to test and characterize, it is impossible to know for sure whether the model is entirely accurate. Therefore, the conclusions of this paper are subject to the assumptions that we, and others, have made. The theme of the paper remains, though, that researchers should apply the objective tests to determine whether a suggested role or policy is specific to MEMStores. As time goes on and our understanding of MEMStore performance becomes more detailed, or as alternative designs appear, we believe it is useful for these tests to be re-applied.

In a few aspects, MEMStore-specific features can provide substantial benefits for well-matched access patterns, beyond the performance and power levels that would be expected from hypothetical fast disks. This paper discusses three specific examples. First, tip-subset parallelism flexibility, created by expected power and component sharing limitations, can be exploited for two-dimensional data structures accessed in both row- and column-order. Second, lack of access-independent motion (e.g., continuous rotation) makes repeated access to the same location much more efficient than in disks, fitting read-modify-write access sequences well. Third, the ratio of access bandwidth to device capacity is almost two orders of magnitude higher than disk drives, making full device scans a more reasonable access pattern.

The rest of this paper is organized as follows. Section 2 overviews MEMS-based storage and related work. Section 3 describes the standard storage interface and how it works for disks. Section 4 explores how key aspects of this interface fit with MEMStore characteristics. Section 5 gives results applying the objective tests to several roles and policies for MEMStores. Section 6 identifies unique features of MEMStores and how they could be exploited for specific application access patterns. Section 7 discusses major assumptions and their impact on the conclusions. Section 8 summarizes the paper.

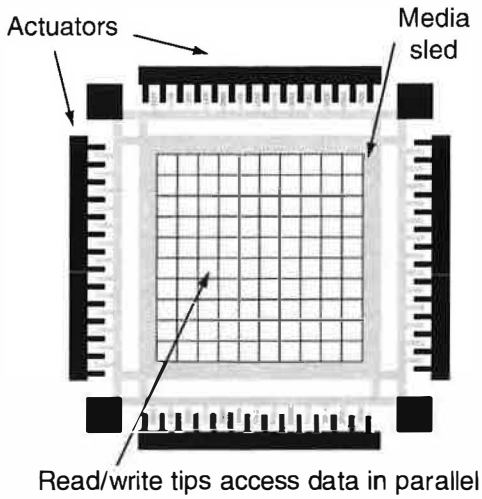


Figure 1: **High-level view of a MEMStore.** The major components of a MEMStore are the sled containing the recording media, MEMS actuators that position the media, and the read/write tips that access the media. The simplified device shown here has a ten by ten array of read/write tips, each of which accesses its own portion (or *square*) of the media. As the media is positioned, each tip accesses the same position within its square, thus providing parallel access to data.

2 Background

MEMStores store data in a very small physical medium that is coated on the surface of a silicon chip. This storage is non-volatile, just as in disk drives. Physically, the devices are much smaller than disks, on the order of a few square centimeters, they store several gigabytes of data, and they access data in a few milliseconds. This section describes in some detail how MEMStores are built, how various designs differ, and what they have in common.

Microelectromechanical systems (MEMS) are microscopic mechanical machines that are fabricated on the surface of silicon chips using techniques similar to those used to make integrated circuits [18]. MEMS devices are used in a wide range of applications, such as accelerometers for automotive airbag systems, high-quality projection systems, and medicine delivery systems. MEMStores use MEMS machinery to position a recording medium and access the data stored in it.

A high-level picture of a MEMStore appears in Figure 1. There are three main components: the media sled, the actuators, and the read/write tips. Data is recorded in

a medium that is coated onto the media sled, so named because it is free to move in two dimensions. It is attached to the chip substrate by beam springs at each corner. The media sled is positioned by a set of actuators, each of which pulls the sled in one dimension. Data is accessed by a set of several thousand read/write tips, which are analogous to the heads of a disk drive.

Accessing data requires two steps. First, the media sled is positioned or "seeks" to the correct offset. Second, the sled moves at a constant rate as the read/write tips access the data stored in the medium. The appropriate subset of tips is engaged to access the desired data.

There are three important differences between the positioning of disk drives and MEMStores. First, the media in the MEMStore can be positioned much more quickly than in a disk because the size, mass, and range of motion of the components are significantly smaller. The seek time of a disk drive averages around 5 ms, while that of a MEMStore is expected to be less than 1 ms. Second, there is no access-independent motion in a MEMStore like the rotation of a disk drive's platters.[1] The rotating media of a disk drive adds, essentially, a random variable (uniform from zero to the full revolution time) that is independent of the access itself to positioning time. Third, positioning takes place in two dimensions.

The last of these differences, that positioning is two-dimensional in nature, is one of the most radical departures of MEMStores from disk drives. Positioning in each dimension takes place independently and in parallel, making the overall positioning time equal to the longer of the two. Once the sled arrives at its destination, there is expected to be a significant settle time while the actuators eliminate oscillations. Section 4.2 discusses the impact of this difference on systems.

2.1 Related work

Building practical MEMStores has been the goal of several major research labs, universities, and startup companies around the world for over a decade. The three most widely publicized are from IBM Research in Zurich, Carnegie Mellon University, and HP Labs. The three designs differ largely in the types of actuators used to position the media and the methods used to record data in the medium. IBM's Millipede designs use electromagnetic motors and a novel thermomechanical recording technique [19, 33]. The device being designed at Carnegie Mellon University uses electrostatic motors for positioning and standard magnetic recording [1, 2]. The Hewlett-Packard Atomic Resolution Storage project utilizes electrostatic stepper motors, phase-change media, and electron beams to record data [12]. Despite these differences, however, each shares the same basic design shown in Figure 1, utilizing a moving media sled and a large array of read/write tips. In the Millipede chip, the read/write tips are in constant physical contact with the media, raising some questions about wear. The others maintain a constant spacing between the tips and the media.

The performance of the various actuator types seems to be similar, but their energy consumption differs somewhat. The electromagnetic actuators of the IBM Millipede chip draw more current, and hence consume more energy, as the media sled is pulled further from its rest position [23, 33]. The electrostatic actuators require higher voltages as the sled is displaced further, but require little current, so the energy consumption is lower overall. This difference could lead to interesting tradeoffs between positioning distance and energy consumption for MEMStores with electromagnetic actuators.

Since MEMStores are still being developed, systems researchers with knowledge of how they may be used can influence their design. Researchers have studied the many physical parameters of MEMStores and how those parameters should be chosen to improve performance on various workloads [7, 17].

Several researchers have studied the various roles that MEMStores may take in computer systems. Schlosser et al. [28] simulated various application workloads on MEMStores, and found that runtime decreased by 1.9–4.4×. They also found that using MEMStores as a disk cache improved I/O response time by up to 3.5×. Hong [13] evaluated using MEMStores as a metadata cache, improving system performance by 28–46% for user workloads. Rangaswami et al. [22] proposed using MEMStores in streaming media servers as buffers between the disks and DRAM. Uysal et al. [32] evaluated the use of MEMStores as components in disk arrays. In evaluating the various roles that a MEMStore may take in a system, it is useful to apply the two objective tests described in Section 1. In this way, one can determine if benefits come from the fact that the workload is particularly well-matched to a MEMStore, or just the fact that a MEMStore is faster than current disks.

Various policies for tailoring access to MEMStores have been suggested. Griffin et al. [11] studied scheduling algorithms, layout schemes, and power management policies, using a disk-like interface. Lin et al. [15] also studied several power conservation strategies for MEMStores. Several groups have suggested MEMStore-specific request scheduling algorithms [14, 36]. These groups have not applied their scheduling algorithms to disk drives to see if they are MEMStore-

[1] Some MEMStore designers have discussed building devices that operate in a resonant mode, in which the media sled moves in resonance along the recording dimension. Such a design would change this assumption and there would be access-independent motion, just like the rotation of the platters in a disk drive.

specific, and we believe it is likely that their algorithms will apply equally well to disks. Lastly, two groups have proposed using tip-subset parallelism in MEMStores to efficiently access tabular data structures [29, 35, 37]. Again, in evaluating potential policies that will be used for MEMStores, one can use the two objective tests to decide whether the policy is MEMStore-specific, or if it can be applied to both MEMStores and disk systems.

3 Standard storage abstractions

High-level storage interfaces (e.g., SCSI and ATA) hide the complexities of mechanical storage devices from the systems that use them, allowing them to be used in a standard, straightforward fashion. Different devices with the same interface can be used without the system needing to change. Also, the system does not need to manage the low-level details of the storage device. Such interfaces are common across a wide variety of storage devices, including disk drives, disk arrays, and FLASH- and RAM-based devices.

Today's storage interface abstracts a storage device as a linear array of fixed-sized *logical blocks* (usually 512 bytes). Details of the mapping of logical blocks to physical media locations are hidden. The interface allows systems to READ and WRITE ranges of blocks by providing a starting logical block number (*LBN*) and a block count.

Unwritten contract: Although no performance specifications of particular access types are given, an unwritten contract exists between host systems and storage devices supporting these standard interfaces (e.g., disks). This unwritten contract has three terms:

- Sequential accesses are best, much better than non-sequential.
- An access to a block near the previous access in *LBN* space is usually considerably more efficient than an access to a block farther away.
- Ranges of the *LBN* space are interchangeable, such that bandwidth and positioning delays are affected by relative *LBN* addresses but not absolute *LBN* addresses.

Application writers and system designers assume the terms of this contract in trying to improve performance.

3.1 Disks and standard abstractions

Disk drives are multi-dimensional machines, with data laid out in concentric circles on one or more media platters that rotate continuously. Data is divided into fixed-sized units, called sectors (usually 512 bytes to match the *LBN* size). The sector (and, thereby, *LBN*) size was originally driven by a desire to amortize both positioning costs and the overhead of the powerful error-correcting codes (ECC) required for robust magnetic data storage. The densities and speeds of today's disk drives would be impossible without these codes, and many disk technologists would like the sector size (and, thus, the *LBN* size) to grow by an order of magnitude to support more powerful codes. Each sector is addressed by a tuple, denoting its cylinder, surface, and rotational position.

*LBN*s are mapped onto the physical sectors of the disk to take advantage of the disk's characteristics. Sequential *LBN*s are mapped to sequential rotational positions within a single track, which leads to the first point of the unwritten contract. Since the disk is continuously rotating, once the heads are positioned, sequential access is very efficient. Non-sequential access incurs large re-positioning delays. Successive tracks of *LBN*s are traditionally mapped to surfaces within cylinders, and then to successive cylinders. This leads to the second point of the unwritten contract: that distant *LBN*s map to distant cylinders, leading to longer seek times.

The linear abstraction works for disk drives, despite their clear three-dimensional nature, because two of the dimensions are largely uncorrelated with *LBN* addressing. Access time is the sum of the time to position the read/write heads to the destination cylinder (seek time), the time for the platters to reach the appropriate rotational offset (rotational latency), and the time to transfer the data to or from the media (transfer time). Seek time and rotational latency usually dominate transfer time. The heads are positioned as a unit by the seek arm, meaning that it usually doesn't matter which surface is being addressed. Unless the abstraction is stripped away, rotational latency is nearly impossible to predict because the platters are continuously rotating and so the starting position is essentially random. The only dimension that remains is that across cylinders, which determines the seek time.

Seek time is almost entirely dependent on the distance traversed, not on the absolute starting and ending points of the seek. This leads to the third point of the unwritten contract. Ten years ago, all disk tracks had the same number of sectors, meaning that streaming bandwidths (and, thus, transfer times) were uniform across the *LBN* space. Today's zoned disk geometries, however, violate the third term since streaming bandwidth varies between zones.

3.2 Holes in the abstraction boundary

Over its fifteen year lifespan, shortcomings of the interface and unwritten contract have been identified. Perhaps the most obvious violation has been the emergence of multi-zone disks, in which the streaming bandwidth varies by over 50% from one part of the disk to another. Some application writers exploit this difference by explicitly using the low-numbered LBNs, which are usually mapped to the outer tracks. Over time, this may

become a fourth term in the unwritten contract.

Some have argued [4, 27] that the storage interface should be extended for disk arrays. Disk arrays contain several disks which are combined to form one or more logical volumes. Each volume can span multiple disks, and each disk may contain parts of multiple volumes. Hiding the boundaries, parallelism, and redundancy schemes prevents applications from exploiting them. Others have argued [8] that, even for disks, the current interface is not sufficient. For example, knowing track boundaries can improve performance for some applications [26].

The interface persists, however, because it greatly simplifies most aspects of incorporating storage components into systems. Before this interface became standard, systems used a variety of per-device interfaces. These were replaced because they complicated systems greatly and made components less interchangeable. This suggests that the bar should be quite high for a new storage component to induce the introduction of a new interface or abstraction.

It is worth noting that some systems usefully throw out abstraction boundaries entirely, and this is as true in storage as elsewhere. In particular, storage researchers have built tools [25, 30] for extracting detailed characteristics of storage devices. Such characteristics have been used for many ends: writing blocks near the disk head [39], reading a replica near the disk head [38], inserting background requests into foreground rotational latencies [16], and achieving semi-preemptible disk I/O [5]. Given their success, adding support for such ends into component implementations or even extending interfaces may be appropriate. But, they do not represent a case for removing the abstractions in general.

4 MEMStores and standard abstractions

Using a standard storage abstraction for MEMStores has the advantage of making them immediately usable by existing systems. Interoperability is important for getting MEMStores into the marketplace, but if the abstractions that are used make performance suffer, then there is reason to consider something different.

This section explains how the details of MEMStore operation make them naturally conform to the storage abstraction used for today's disks. Also, the unwritten contract that applications expect will remain largely intact.

4.1 Access method

The standard storage interface allows accesses (READs and WRITEs) to ranges of sizeable fixed-sized blocks. The question we ask first is whether such an access method is appropriate for a MEMStore.

Is a 512 byte block appropriate, or should the abstraction use something else? It is true that MEMStores can dynamically choose subsets of read/write tips to engage when accessing data, and that these subsets can, in theory, be arbitrarily-sized. However, enough data must be read or written for error-correcting codes (ECC) to be effective. The use of ECC enables high storage density by relaxing error-rate constraints. Since the density of a MEMStore is expected to equal or exceed that of disk drives, the ECC protections needed will be comparable. Therefore, block sizes of the same order of magnitude as disks should be expected. Also, any block's size must be fixed, since it must be read or written in its entirety, along with the associated ECC. Accessing less than a full block, e.g., to save energy [15], would not be possible. The flexibility of being able to engage arbitrary sets of read/write tips can still be used to selectively choose sets of these fixed-sized blocks.

Large block sizes are also motivated by embedded servo mechanisms, coding for signal processing, and the relatively low per-tip data rate of around 1 Mbit/s. The latter means that data will have to be spread across multiple parallel-operating read/write tips to achieve an aggregate bandwidth that is on-par with that of disk drives. Spreading data across multiple read/write tips also introduces physical redundancy that will allow for better tolerance of tip failures. MEMStores will use embedded servo [31], requiring that several bits containing position information be read before any access in order to ensure that the media sled is positioned correctly. Magnetic recording techniques commonly use transitions between bits rather than the bits themselves to represent data, meaning that a sequence of bits must be accessed together. Further, signal encodings use multi-bit codewords that map a sequence of bits to values with interpretable patterns (e.g., not all ones or all zeros). The result is that, in order to access any data after a seek, some amount of data (10 bits in our model) must be read for servo information, and then bits must be accessed sequentially with some coding overhead (10 bits per byte in our model). Given these overheads, a large block size should be used to amortize the costs. This block will be spread across multiple read/write tips to improve data rates and fault tolerance.

Using current storage interfaces, applications can only request ranges of sequential blocks. Such access is reasonable for MEMStores, since blocks are laid out sequentially, and their abstraction should support the same style of access. There may be utility in extending the abstraction to allow applications to request batches of non-contiguous *LBN*s that can be accessed by parallel read/write tips. An extension like this is discussed in Section 6.

4.2 Unwritten contract

Assuming that MEMStore access uses the standard storage interface, the next step is to see if the unwritten contract for disks still holds. If it does, then MEMStores can be used effectively by systems simply as fast disks.

The first term of the unwritten contract is that sequential access is more efficient than random access. This will continue to be the case for MEMStores because data still must be accessed in a linear fashion. The signal processing techniques that are commonly used in magnetic storage are based on transitions between bits, rather than the state of the bits in isolation. Moreover, they only work properly when state transitions come frequently enough to ensure clock synchronization so they encode multi-bit data sequences into alternate codewords. These characteristics dictate that the bits must be accessed sequentially. Designs based on recording techniques other than magnetic will, most likely, encode data similarly. Once the media sled is in motion, it is most efficient for it to stay in motion, so the most efficient thing to access is the next sequential data, just as it is for disks.

The second term of the unwritten contract is that the difference between two *LBN* numbers maps well to the physical distance between them. This is dependent on how *LBN*s are mapped to the physical media, and this mapping can easily be constructed in a MEMStore to make the second point of the unwritten contract be true. A MEMStore is a multi-dimensional machine, just like a disk, but the dimensions are correlated differently. Each media position is identified by a tuple of the X position, the Y position, and the set of read/write tips that are enabled, much like the cylinder/head/rotational position tuples in disks. There are thousands of read/write tips in a MEMStore, and each one accesses its own small portion of the media. Just as the heads in a disk drive are positioned as a unit to the same cylinder, the read/write tips in a MEMStore are always positioned to the same offset within their own portion of the media. The choice of which read/write tips to activate has no correlation with access time, since any set can be chosen for the same cost once the media is positioned.

As with disks, seek time for a MEMStore is a function of seek distance. Since the actuators on each axis are independent, the overall seek time is the maximum of the individual seek times in each dimension, X and Y. But, the X seek time almost always dominates the Y seek time because extra settle time must be included for X seeks, but not for Y seeks. The reason for this is that post-seek oscillations in the X dimension lead to off-track interference, while the same oscillations in the Y dimension affect only the bit rate of the data transfer. Since the overall seek time is the maximum of the two individual seek times, and the X seek time is almost always greater than the Y seek time, the overall seek distance is (almost) uncorrelated with the Y position, as seen in Figure 2. In the end, despite the fact that a MEMStore has multiple dimensions over which to position, the overall access time is (almost) only correlated with just a single dimension, which makes a linear abstraction sufficient.

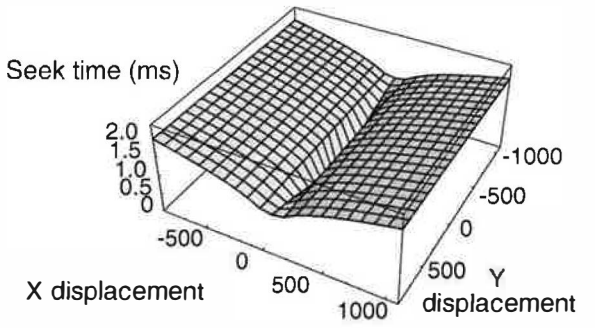


Figure 2: **MEMStore seek curve.** The seek time of a MEMStore is largely uncorrelated with the displacement in the Y dimension due to a large settling time required for the X dimension seek that is not required for the Y dimension seek [7, 10]. The overall seek time is the maximum of the two independent seek times.

The last term of the unwritten contract states that the *LBN* space is uniform, and that access time does not vary across the range of the *LBN*s. The springs that attach the media sled to the chip do affect seek times by applying a greater restoring force when they are displaced further. However, the effect is minimal, with seek times varying by at most 10–15%, meaning that overall access times at the application level would vary by far less. Also, MEMStores do not need zoned recording. It is safe to say that the last point of the unwritten contract still holds: ranges of the *LBN* space of a MEMStore are interchangeable.

4.3 Possible exceptions

This section has explained how MEMStores fit the same assumptions that make storage abstractions work for disks. There are a few aspects of MEMStores, discussed in Section 6, that set them apart from disks for specific access patterns. These exceptions can be exploited with little or no change to the existing storage interface. Of course, the discussion above is based on current MEMStore designs. Section 7 discusses the most significant design assumptions and what removing them would change.

5 Experiments

There are two objective tests that one should consider when evaluating whether a potential role or policy for MEMStores requires a new abstraction. The *specificity test* asks whether the role or policy is truly MEMStore-specific. The test here is to evaluate the role or policy

Capacity	3.46 GB
Average access time	0.88 ms
Streaming bandwidth	76 MB/s

Table 1: **G2 MEMStore parameters.** These parameters are for the G2 MEMStore design from [28].

Capacity	41.6 GB
Rotation speed	55,000 RPM
One-cylinder seek time	0.1 ms
Full-stroke seek time	2.0 ms
Head switch time	0.01 ms
Number of cylinders	39511
Number of surfaces	2
Average access time	0.88 ms
Streaming bandwidth	100 MB/s

Table 2: **Überdisk parameters.** The Überdisk is a hypothetical future disk drive. Its parameters are scaled from current disks, and are meant to represent those of a disk that matches the performance of a MEMStore. The average response time is for a random workload which exercised only the first 3.46 GB of the disk in order to match the capacity of the G2 MEMStore.

for both a MEMStore and a (hypothetical) disk drive of equivalent performance. If the benefit is the same, then the role or policy (however effective) is not truly MEMStore-specific. Given that the role or policy passes the specificity test, the *merit test* determines whether the difference makes a significant-enough impact in performance (or whatever metric) to justify customizing the system. This section examines a potential role and a potential MEMStore-specific policy, under the scrutiny of these two tests.

5.1 G2 MEMStore

The MEMStore that we use for evaluation is the G2 model from [28]. Its basic parameters are given in Table 1. We use DiskSim, a freely-available storage system simulator, to simulate the MEMStore [6].

5.2 Überdisk: A hypothetical fast disk

For comparison, we use a hypothetical disk design, which we call the Überdisk, that approximates the performance of a G2 MEMStore. Its parameters are based on extrapolating from today's disk characteristics, and are given in Table 2. The Überdisk is also modeled using DiskSim. In order to do a capacity-to-capacity comparison, we use only the first 3.46 GB of the Überdisk to match the capacity of the G2 MEMStore. The two devices have equivalent performance under a random workload of 4 KB requests that are uniformly distributed across the capacity (3.46 GB) and arrive one at a time. Since our model of MEMStores does not include a cache, we disabled the cache of the Überdisk.

We based the seek curve on the formula from [24], choosing specific values for the one-cylinder and full-stroke seeks. Head switch and one-cylinder seek times are expected to decrease in the future due to microactuators integrated into disk heads, leading to shorter settle times. With increasing track densities, the number of platters in disk drives is decreasing steadily, so the Überdisk has only two surfaces. The zoning geometry is based on simple extrapolation of current linear densities.

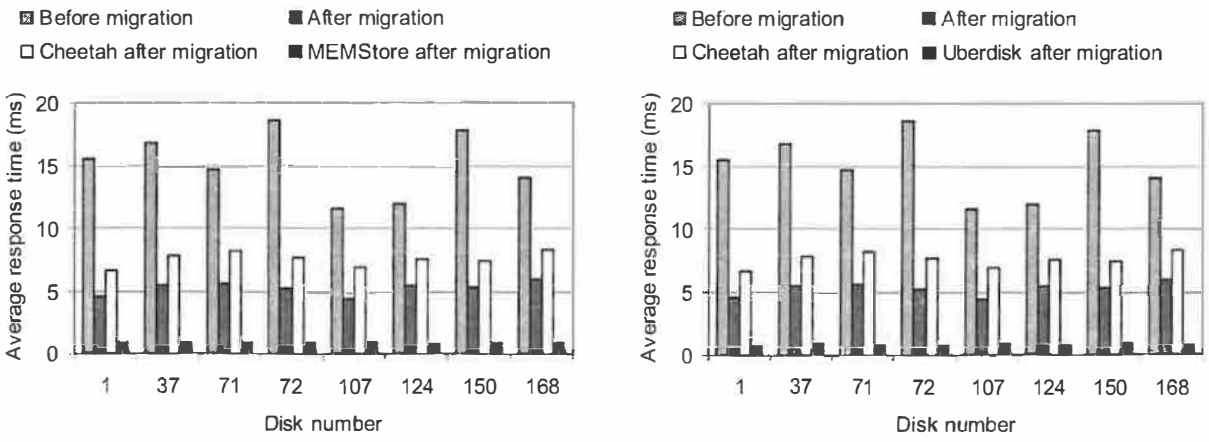
An Überdisk does not necessarily represent a realistic disk; for example, a rotation rate of 55,000 RPM (approximately twice the speed of a dental drill) may never be attainable in a reasonably-priced disk drive. However, this rate was necessary to achieve an average rotational latency that is small enough to match the average access time of the MEMStore. The Überdisk is meant to represent the parameters that would be required of a disk in order to match the performance of a MEMStore. If the performance of a workload running on a MEMStore is the same as it running on an Überdisk, then we can say that any performance increase is due only to the intrinsic speed of the device, and not due to the fact that it is a MEMStore or an Überdisk. If the performance of the workload differs on the two devices, then it must be especially well-matched to the characteristics of one device or the other.

5.3 Role: MEMStores in disk arrays

One of the roles that has been suggested for MEMStores in systems is that of augmenting or replacing some or all of the disks in a disk array to increase performance [28, 32]. However, the lower capacity and potentially higher cost of MEMStores suggest that it would be impractical to simply replace all of the disks. Therefore, they represent a new tier in the traditional storage hierarchy, and it will be important to choose which data in the array to place on the MEMStores and which to store on the disks. Uysal et al. evaluate several methods for partitioning data between the disks and the MEMStores in a disk array [32]. We describe a similar experiment below, in which a subset of the data stored on the back-end disks in a disk array is moved to a MEMStore.

We can expect some increase in performance from doing this, as Uysal et al. report. However, our question here is whether the benefits are from a MEMStore-specific attribute, or just from the fact that MEMStores are faster than the disks used in the disk array. To answer this question, we apply the specificity test by comparing the performance of a disk array back-end workload on three storage configurations. The first configuration uses just the disks that were originally in the disk array. The second configuration augments the overloaded disks with a MEMStore. The third does the same with an Überdisk.

The workload is a disk trace gathered from the disks



(a) MEMStore (b) Überdisk

Figure 3: **Using MEMStores in a disk array.** These graphs show the result of augmenting overloaded disks in a disk array with faster storage components: a MEMStore (a) or an Überdisk (b). In both cases, the busiest logical volume on the original disk (a 73 GB Seagate Cheetah) is moved to the faster device. Requests to the busiest logical volume are serviced by the faster device, and the traffic to the Cheetah is reduced. The results for both experiments are nearly identical, leading to the conclusion that the MEMStore and theÜberdisk are interchangeable in this role (e.g., it is not MEMStore-specific.)

in the back-end of an EMC Symmetrix disk array during the summer of 2001. The disk array contained 282 Seagate Cheetah 73 GB disk drives, model number ST173404. From those, we have chosen the eight busiest (disks 1, 37, 71, 72, 107, 124, 150, and 168), which have an average request arrival rate of over 69 requests per second for the duration of the trace, which was 12.5 minutes. Each disk is divided into 7 logical volumes, each of which is approximately 10 GB in size. For each "augmented" disk, we move the busiest logical volume to a faster device, either a MEMStore or an Überdisk. The benefit should be twofold: first, response times for the busiest logical volume will be improved, and second, traffic to the original disk will be reduced. Requests to the busiest logical volume are serviced by the faster device (either a MEMStore or an Überdisk), and all other requests are serviced by the original Cheetah disk.

Figure 3(a) shows the result of the experiment with the MEMStore. For each disk, the first bar shows the average response time of the trace running just on the Cheetah, which is 15.1 ms across all of the disks. The second bar shows the average response time of the same requests after the busiest logical volume has been moved to the MEMStore. Across all disks, the average is now 5.24 ms. The third and fourth bars show, respectively, the average response time of the Cheetah with the reduced traffic after augmentation, and the average response time of the busiest logical volume, which is now stored on the MEMStore. We indeed see the benefits anticipated — the average response time of requests to the busiest logical volume have been reduced to 0.86 ms, and the reduction of load on the Cheetah disk has resulted in a lower average response time of 7.56 ms.

Figure 3(b) shows the same experiment, but with the busy logical volume moved to an Überdisk rather than a MEMStore. The results are almost exactly the same, with the response time of the busiest logical volume migrated to the Überdisk being around 0.84 ms, and the overall response time reduced from 15.1 ms to 5.21 ms.

The fact that the MEMStore and the Überdisk provide the same benefit in this role means that it fails the specificity test. In this role, a MEMStore really can be considered to be just a fast disk. The workload is not specifically matched to the use of a MEMStore or an Überdisk, but can clearly be improved with the use of any faster device, regardless of its technology.

Although it is imperceptible in Figure 3, the Überdisk gives slightly better performance than the MEMStore because it benefits more from workload locality due to the profile of its seek curve. The settling time in the MEMStore model makes any seek expensive, with a gradual increase up to the full-stroke seek. The settling time of the Überdisk is somewhat less, leading to less expensive initial seek and a steeper slope in the seek curve up to the full-stroke seek. The random workload we used to compare devices has no locality, but the disk array trace does.

To explore this further, we re-ran the experiment with two other disk models, which we call Simpledisk-constant and Simpledisk-linear. Simpledisk-constant responds to requests in a fixed amount of time, equal to

that of the response time of the G2 MEMStore under the random workload: 0.88 ms. The response time of Simpledisk-linear is a linear function of the distance from the last request in *LBN* space. The endpoints of the function are equal to the single-cylinder and full-stroke seek times of the Überdisk, which are 0.1 ms and 2.0 ms, respectively. Simpledisk-constant should not benefit from locality, and Simpledisk-linear should benefit from locality even more than either the MEMStore or the Überdisk. Augmenting the disk array with these devices gives response times to the busiest logical volume of 0.92 ms and 0.52 ms, respectively. As expected, Simpledisk-constant does not benefit from workload locality and Simpledisk-linear benefits more than a real disk.

Uysal proposed several other MEMStore/disk combinations in [32], including replacing all of the disks with MEMStores, replacing half of the mirrors in a mirrored configuration, and using the MEMStore as a replacement of the NVRAM cache. In all of these cases, and in most of the other roles outlined in Section 2.1, the MEMStore is used simply as a block store, with no tailoring of access to MEMStore-specific attributes. We believe that if the specificity test were applied, and an Überdisk was used in each of these roles, the same performance improvement would result. Thus, the results of prior research apply more generally to faster mechanical devices.

5.4 Policy: distance-based scheduler

Mechanical and structural differences between MEMStores and disks suggest that request scheduling policies that are tailored to MEMStores may provide better performance than ones that were designed for disks. Upon close examination, however, the physical and mechanical motions that dictate how a scheduler may perform on a given device continue to apply to MEMStores as they apply to disks. This may be surprising at first glance, since the devices are so different, but after examining the fundamental assumptions that make schedulers work for disks, it is clear that those assumptions are also true for MEMStores.

To illustrate, we give results for a MEMStore-specific scheduling algorithm called *shortest-distance-first*, or SDF. Given a queue of requests, the algorithm compares the Euclidean distance between the media sled's current position and the offset of each request and schedules the request that is closest. The goal is to exploit a clear difference between MEMStores and disks: that MEMStores position over two dimensions rather than only one. When considering the specificity test, it is not surprising that this qualifies as a MEMStore-specific policy. Disk drives do, in fact, position over multiple dimensions, but predicting the positioning time based on any dimension other than the cylinder distance is very difficult outside of disk firmware. SDF scheduling for MEMStores is easier and could be done outside of the device firmware, since it is only based on the logical-to-physical mapping of the device's sectors and any defect management that is used, assuming that the proper geometry information is exposed through the MEMStore's interface.

The experiment uses a random workload of 250,000 requests uniformly distributed across the capacity of the MEMStore. Each request had a size of 8 KB. This workload is the same as that used in [34] to compare request scheduling algorithms. The experiment tests the effectiveness of the various algorithms by increasing the arrival rate of requests until saturation — the point at which response time increases dramatically because the device can no longer service requests fast enough and the queue grows without bound.

The algorithms compared were first-come-first-served (FCFS), cyclic LOOK (CLOOK), shortest-seek-time-first (SSTF), shortest-positioning-time-first (SPTF), and shortest-distance-first (SDF). The first three are standard disk request schedulers for use in host operating systems. FCFS is the baseline for comparison, and is expected to have the worst performance. CLOOK and SSTF base their scheduling decisions purely on the *LBN* number of the requests, utilizing the unwritten assumption that *LBN* numbers roughly correspond to physical positions [34]. SPTF uses a model of the storage device to predict service times for each request, and can be expected to give the best performance. The use of the model by SPTF breaks the abstraction boundaries because it provides the application with complete details of the device parameters and operation. The SDF scheduler requires the capability to map *LBN* numbers to physical locations, which breaks the abstraction, but does not require detailed modeling, making it practical to implement in a host OS.

Figure 4 shows the results. As expected, FCFS and SPTF perform the worst and the best, respectively. CLOOK and SSTF don't perform as well as SPTF because they use only the *LBN* numbers to make scheduling decisions. The SDF scheduler performs slightly worse than CLOOK and SSTF. The reason is that positioning time is not as well correlated with two-dimensional position, as described in Section 4.2. As such, considering the two-dimensional seek distance does not provide any more utility than just considering the one-dimensional seek distance, as CLOOK and SSTF effectively do. Thus, the suggested policy fails the merit test: the same or greater benefit can be had with existing schedulers that don't need MEMStore-specific knowledge. This is based, of course, on the assumption that settling time is a significant component of position-

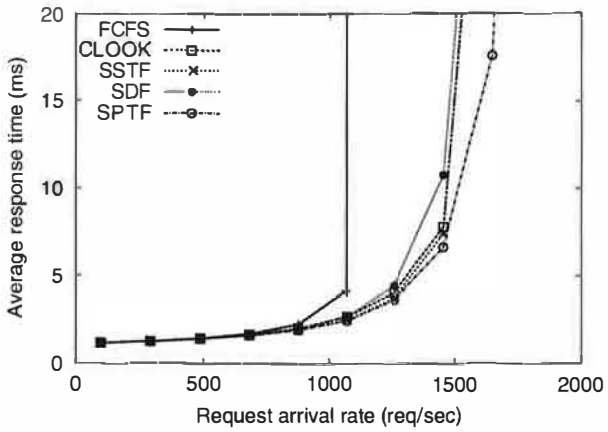


Figure 4: **Performance of shortest-distance-first scheduler.** A MEMStore-specific scheduler that accounts for two-dimensional position gives no benefit over simple schedulers that use a linear abstraction (CLOOK and SSTF). This is because seek time in a MEMStore is correlated most strongly with only distance in the X dimension.

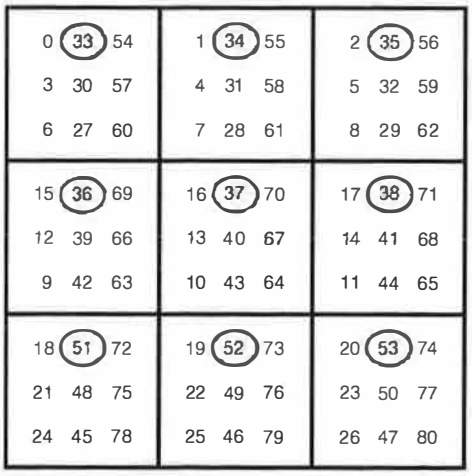


Figure 5: **Data layout with a set of equivalent *LBN*s highlighted.** The *LBN*s marked with ovals are at the same location within each square and, thus, are "equivalent". That is, they can potentially be accessed in parallel.

ing time. Section 7 discusses the effect of removing this assumption.

The fundamental reason that scheduling algorithms developed for disks work well for MEMStores are that seek time is strongly dependent on seek distance, but only the seek distance in a single dimension. The seek time is only correlated to a single dimension, which is exposed by the linear abstraction. The same is true for disks when one cannot predict the rotational latencies, in which only the distance that the heads must move across cylinders is relevant. Hence, a linear logical abstraction is as justified for MEMStores as it is for disks.

Of course, there may be yet-unknown policies that exploit features that are specific to MEMStores, and we expect research to continue in this area. When considering potential policies for MEMStores, it is important to keep the two objective tests in mind. In particular, these tests can expose a lack of need for a new policy or, better yet, the fact that the policy is equally applicable to disks and other mechanical devices.

6 MEMStore-specific features

This section describes three MEMStore-specific features that clearly set them apart from disks, offering significant performance improvements for well-matched workloads. Exploiting such features may require a new abstraction or, at least, changes in the unwritten contract between systems and storage.

6.1 Tip-subset parallelism

MEMStores have an interesting access parallelism feature that does not exist in modern disk drives. Specifically, subsets of a MEMStore's thousands of read/write tips can be used in parallel, and the particular subset can be dynamically chosen. This section briefly describes how such access parallelism can be exposed to system software, with minimal extensions to the storage interface, and utilized cleanly by applications. Interestingly, our recent work [27] has shown the value of the same interface extensions for disk arrays, suggesting that this is a generally useful storage interface change.

Figure 5 shows a simple MEMStore with nine read/write tips and nine sectors per tip. Each read/write tip addresses its own section of the media, denoted by the nine squares in the figure. Sectors that are at the same physical offset within each square, such as those indicated with ovals, are addressed simultaneously by the tip array. We call these sectors *equivalent*, because they can be accessed in parallel. But, in many designs, not all of the tips can be actively transferring data at the same time due to power consumption or component sharing constraints. Using a simple API, an application or OS module could identify sets of sectors that are equivalent, and then choose subsets to access together. Since the *LBN*s which will be accessed together will not fall into a contiguous range, the system will need to be able to request batches of non-contiguous *LBN*s, rather than ranges.

6.1.1 Efficient 2D data structure access

The standard interface forces applications to map their data into a linear address space. For most applications, this is fine. However, applications that use two-dimensional data structures, such as non-sparse matrices or relational database tables, are forced to serialize their storage in this linear address space, making efficient access possible along only one dimension of the data structure. For example, a database can choose to

store its table in column-major order, making column accesses sequential and efficient [3]. Once this choice is made, however, accessing the table in row-major order is very expensive, requiring a full scan of the table to read a single row. One option for making operations in both dimensions efficient is to create two copies of a table; one copy is optimized for column-major access and the other is optimized for row-major access [21]. This scheme, however, doubles the capacity needed for the database and requires that updates propagate to both copies.

With proper allocation of data to a MEMStore *LBN* space, parallel read/write tips can be used to access a table in either row- or column-major order at full speed [29, 35]. The table is arranged such that the same attributes of successive records are stored in sequential *LBN*s. Then, the other attributes of those records are stored in *LBN*s that are equivalent to the original *LBN*s, as in Figure 5. This layout preserves the two-dimensionality of the original table on the physical media of the MEMStore. Then, when accessing the data, the media sled is positioned and the appropriate read/write tips are activated to read data either in row- or column-major order.

To quantify the advantages of such a MEMStore-specific scan operator, we compare the times required for different table accesses. We contrast their respective performance under two different layouts on a single G2 MEMStore device. The first layout, called *normal*, is the traditional row-major access optimized page layout used in almost all database systems [20].

The second layout, called *MEMStore*, uses the MEMStore-specific layout and access described above. The sample database table consists of 4 attributes a_1, a_2, a_3, and a_4 sized at 8, 32, 15, and 16 bytes respectively. The *normal* layout consists of 8 KB pages that hold 115 records. The table size is 10,000,000 records for a total of 694 MB of data.

Figure 6 compares the time of a full table scan for all attributes with four scans of the individual attributes. The total runtime of four single-attribute scans in the *MEMStore* case takes the same amount of time as the full table scan. In contrast, with the *normal* layout, the four successive scans take four times as long as the full table scan. Most importantly, a scan of a single attribute in the *MEMStore* case takes only the amount of time needed for a full-speed scan of the corresponding amount of data, since all of the available read/write tips read records of the one attribute. This result represents a compelling performance improvement over current database systems. This policy for MEMStores passes both the specificity test and the merit test.

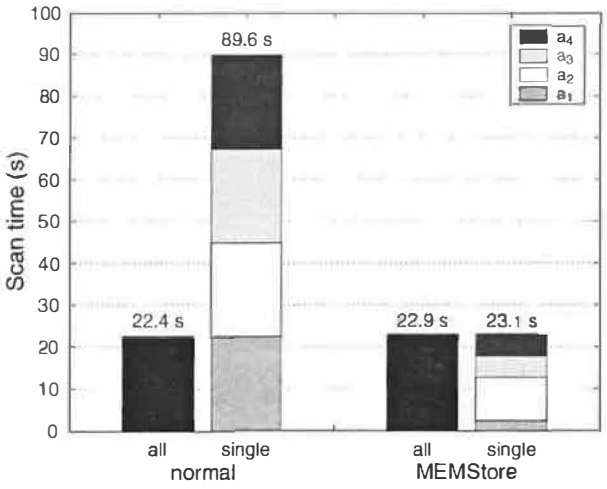


Figure 6: **Database table scan with different number of attributes.** This graph shows the runtime of scanning 10,000,000 records using a G2 MEMStore. For each of the two layouts, the left bar, labeled "all," shows the time to scan the entire table with four attributes. The right bar, labeled "single," is composed of four separate scans of each successive attribute, simulating the situation where multiple queries access different attributes. Since the *MEMStore* layout takes advantage of MEMStore's tip-subset parallelism, each attribute scan runtime is proportional to the amount of data occupied by that attribute. The *normal* layout, on the other hand, must read the entire table to fetch any one attribute.

6.2 Quick turnarounds

Another aspect of MEMStores that differs from disk drives is their ability to quickly access an *LBN* repeatedly. In a disk, repeated reads to an *LBN* may be serviced from the disk's buffer, but repeated synchronous writes or read/modify/write sequences will incur a full rotation, 4-8 ms on most disks, for each access. A MEMStore, however, can simply change the direction that the media sled is moving, which is predicted to take less than a tenth of a millisecond [11]. Read/modify/write sequences are prevalent in parity-based redundancy schemes, such as RAID-5, in which the old data and parity must each be read and then updated for each single block write. Repeated synchronous writes are common in database log files, where each commit entry must propagate to disk. Such operations are much more expensive in a disk drive.

6.3 Device scan time

Although the volumetric density of MEMStores is on-par with that of disk drives, the per-device capacity is much less. For example, imagine two 100 GB "storage bricks," one using disk storage and the other using MEMStores. Given that the volumetric densities are equal, the two bricks would consume about the same amount of physical volume. But, the MEMStore brick

would require at least ten devices, while the disk-based brick could consist of just one device. This means that the MEMStore-based brick would have more independent actuators for accessing the data, leading to several interesting facts. First, the MEMStore-based brick could handle more concurrency, just as in a disk array. Second, MEMStores in the brick that are idle could be turned off while others in the brick are still servicing requests, reducing energy consumption. Third, the overall time to scan the entire brick could be reduced, since some (or all) of the devices could access data in parallel. This assumes that the bus connecting the brick to the system is not a bottleneck, or that the data being scanned is consumed within the brick itself. The lower device scan time is particularly interesting because disk storage is becoming less accessible as device capacities grow more quickly than access speeds [9].

Simply comparing the time to scan a device in its entirety, a MEMStore could scan its entire capacity in less time than a single disk drive. At 100 MB/s, a 10 GB MEMStore is scanned in only 100 s, while a 72 GB disk drive takes 720 s. As a result, strategies that require entire-device scans, such as scrubbing or virus scanning, become much more feasible.

7 Major assumptions

Unfortunately, MEMStores do not exist yet, so there are no prototypes that we can experiment with, and they are not expected to exist for several more years. As such, we must base all experiments on simulation and modeling. We have based our models on detailed discussions with researchers who are designing and building MEMStores, and on an extensive study of the literature. The work and the conclusions in this paper are based on this modeling effort, and is subject to its assumptions about the devices. This section outlines two of the major assumptions of the designers and how our conclusions would change given different assumptions.

Some of our conclusions are based on the assumption that post-seek settling time will affect one seek dimension more than the other. This effectively uncorrelates seek time with one of the two dimensions, as described in Section 4.2. The assumption is based on the observation that different mechanisms determine the settling time in each of the two axes, X and Y. Settling time is needed to damp oscillations enough for the read/write tips to reliably access data. In all published MEMStore designs, data is laid out linearly along the Y-axis, meaning that oscillations in Y will appear to the channel as minor variations in the data rate. Contrast this with oscillations in the X-axis, which pull the read/write tips off-track. Because one axis is more sensitive to oscillation than the other, its positioning delays will dominate the other's, unless the oscillations can be damped in near-zero time.

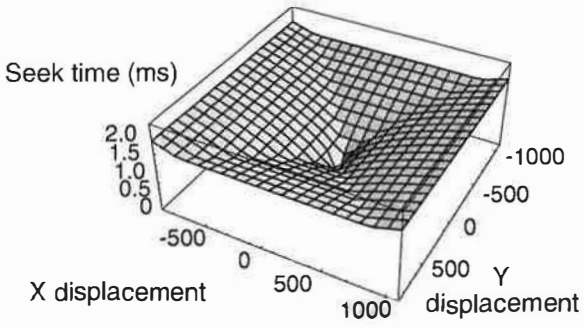


Figure 7: **MEMStore seek curve without settling time.** Without the settling time, the seek curve of a MEMStore is strongly correlated with displacement in both dimensions [10].

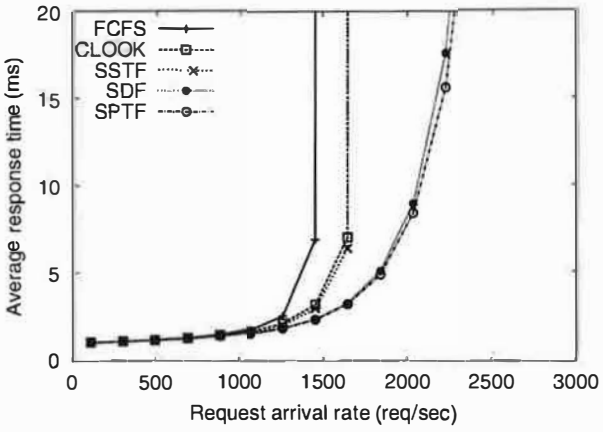


Figure 8: **Performance of shortest-distance-first scheduler without settle time.** If post-seek settle time is eliminated, then the seek time of a MEMStore becomes strongly correlated with both the X and Y positions. In this case, a scheduler that takes into account both dimensions provides much better performance than those that only consider a single dimension (CLOOK and SSTF).

If this assumption no longer held, and oscillations affected each axis equally, then MEMStore-specific policies that take into account the resulting two-dimensionality of the seek profile, as illustrated in Figure 7, would become more valuable. Now, for example, two-dimensional distance would be a much better predictor of overall positioning time. Figure 8 shows the result of repeating the experiment from Section 5.4, but with the post-seek settle time set to zero. In this case, the performance of the SDF scheduler very closely tracks shortest-positioning-time-first, SPTF, the scheduler based on full knowledge of positioning time. Further, the difference between SDF and the two algorithms based on single-dimension position (CLOOK and SSTF) is now very large. CLOOK and SSTF have worse performance because they ignore the second dimension that is now correlated strongly with positioning time.

Another closely-related assumption is that data in a MEMStore is accessed sequentially in a single dimension. One could imagine a MEMStore in which data is accessed one point at a time. As a simple example, imagine that the media sled would move to a single position and then engage 8×512 read/write probes (plus ECC tips) in parallel to read one 512 byte sector from the media at once. From that point, the media sled could then re-position in either the X or Y dimension and read another 512 byte sector. In fact, the device could stream sequentially along either dimension. Current designs envision using embedded servo to keep the read/write tips on track, just as in disks [31]. Both servo and codewords would have to be encoded along both dimensions somehow to allow streaming along either. The ability to read sequentially along either dimension at an equal rate could improve the performance of applications using two-dimensional data structures, as described in Section 6.1.1. Rather than using tip subset parallelism, data tables could be stored directly in their original format on the MEMStore, and then accessed in either direction efficiently. Note, however, that the added complexity of the coding and access mechanisms would be substantial, making this unlikely to occur.

8 Summary

One question that should be asked when considering how to use MEMStores in computer systems is whether they have unique characteristics that should be exploited by systems, or if they can be viewed as small, low-power, fast disk drives. This paper examines this question by establishing two objective tests that can be used to identify the existence and importance of relevant MEMStore-specific features. If an application utilizes a MEMStore-specific feature, then there may be reason to use something other than existing disk-based abstractions. After studying the fundamental reasons that the existing abstraction works for disks, we conclude that the same reasons hold true for MEMStores, and that a disk-like view is justified. Several case studies of potential roles that MEMStores may take in systems and policies for their use support this conclusion.

Acknowledgements

We thank the members of the MEMS Laboratory at CMU for helping us understand the technology of MEMStores. We would like to thank the anonymous reviewers and our shepherd, David Patterson, for helping to improve this paper. We thank the members and companies of the PDL Consortium (EMC, Hewlett-Packard, Hitachi, IBM, Intel, LSI Logic, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. This work is funded in part by NSF grant CCR-0113660.

References

[1] L. R. Carley, J. A. Bain, G. K. Fedder, D. W. Greve, D. F. Guillou, M. S. C. Lu, T. Mukherjee, S. Santhanam, L. Abelmann, and S. Min. Single-chip computers with microelectromechanical systems-based magnetic memory. *Journal of Applied Physics*, **87**(9):6680–6685, 1 May 2000.

[2] Center for Highly Integrated Information Processing and Storage Systems, Carnegie Mellon University. http://-www.ece.cmu.edu/research/chips/.

[3] G. P. Copeland and S. Khoshafian. A decomposition storage model. *ACM SIGMOD International Conference on Management of Data* (Austin, TX, 28–31 May 1985), pages 268–279. ACM Press, 1985.

[4] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. *Summer USENIX Technical Conference* (Monterey, CA, 10–15 June 2002), pages 177–190, 2002.

[5] Z. Dimitrijević, R. Rangaswami, and E. Chang. Design and implementation of semi-preemptible IO. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 145–158. USENIX Association, 2003.

[6] The DiskSim Simulation Environment (Version 3.0). http://www.pdl.cmu.edu/DiskSim/index.html.

[7] I. Dramaliev and T. M. Madhyastha. Optimizing probe-based storage. *Conference on File and Storage Technologies* (San Francisco, CA, 31–02 April 2003), pages 103–114. USENIX Association, 2003.

[8] G. R. Ganger. *Blurring the line between OSs and storage devices.* Technical report CMU–CS–01–166. Carnegie Mellon University, December 2001.

[9] J. Gray. A conversation with Jim Gray. *ACM Queue*, **1**(4). ACM, June 2003.

[10] J. L. Griffin, S. W. Schlosser, G. R. Ganger, and D. F. Nagle. Modeling and performance of MEMS-based storage devices. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, 17–21 June 2000). Published as *Performance Evaluation Review*, **28**(1):56–65, 2000.

[11] J. L. Griffin, S. W. Schlosser, G. R. Ganger, and D. F. Nagle. Operating system management of MEMS-based storage devices. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 227–242. USENIX Association, 2000.

[12] Hewlett-Packard Laboratories Atomic Resolution Storage. http://www.hpl.hp.com/research/storage.html.

[13] B. Hong. Exploring the usage of MEMS-based storage as metadata storage and disk cache in storage hierarchy. http://www.cse.ucsc.edu/~hongbo/publications/-mems-metadata.pdf.

[14] B. Hong, S. A. Brandt, D. D. E. Long, E. L. Miller, K. A. Glocer, and Z. N. J. Peterson. Zone-based shortest positioning time first scheduling for MEMS-based storage devices. *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems* (Orlando, FL, 12–15 October 2003), 2003.

[15] Y. Lin, S. A. Brandt, D. D. E. Long, and E. L. Miller. Power conservation strategies for MEMS-based storage devices. *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems* (Fort Worth, TX, October 2002), 2002.

[16] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 275–288. USENIX Association, 2002.

[17] T. M. Madhyastha and K. P. Yang. Physical modeling of probe-based storage. *IEEE Symposium on Mass Storage Systems* (April 2001). IEEE, 2001.

[18] N. Maluf. *An introduction to microelectromechanical systems engineering.* Artech House, 2000.

[19] The Millipede: A future AFM-based data storage system. http://www.zurich.ibm.com/st/storage/millipede.html.

[20] R. Ramakrishnan and J. Gehrke. *Database management systems*, number 3rd edition. McGraw-Hill, 2003.

[21] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. *International Conference on Very Large Databases* (Hong Kong, China, 20–23 August 2002), pages 430–441. Morgan Kaufmann Publishers, Inc., 2002.

[22] R. Rangaswami, Z. Dimitrijević, E. Chang, and K. E. Schauser. MEMS-based disk buffer for streaming media servers. *International Conference on Data Engineering* (Bangalore, India, 05–08 March 2003), 2003.

[23] H. Rothuizen, U. Drechsler, G. Genolet, W. Häberle, M. Lutwyche, R. Stutz, R. Widmer, and P. Vettiger. Fabrication of a micromachined magnetic X/Y/Z scanner for parallel scanning probe applications. *Microelectronic Engineering*, **53**:509–512, June 2000.

[24] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994.

[25] J. Schindler and G. R. Ganger. *Automated disk drive characterization.* Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[26] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 259–274. USENIX Association, 2002.

[27] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger. Atropos: A disk array volume manager for orchestrated use of disks. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004). USENIX Association, 2004.

[28] S. W. Schlosser, J. L. Griffin, D. F. Nagle, and G. R. Ganger. Designing computer systems with MEMS-based storage. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 12–15 November 2000). Published as *Operating Systems Review*, **34**(5):1–12, 2000.

[29] S. W. Schlosser, J. Schindler, A. Ailamaki, and G. R. Ganger. *Exposing and exploiting internal parallelism in MEMS-based storage.* Technical Report CMU–CS–03–125. Carnegie-Mellon University, Pittsburgh, PA, March 2003.

[30] N. Talagala, R. H. Dusseau, and D. Patterson. *Microbenchmark-based extraction of local and global disk characteristics.* Technical report CSD–99–1063. University of California at Berkeley, 13 June 2000.

[31] B. D. Terris, S. A. Rishton, H. J. Mamin, R. P. Ried, and D. Rugar. Atomic force microscope-based data storage: track servo and wear study. *Applied Physics A*, **66**:S809 S813, 1998.

[32] M. Uysal, A. Merchant, and G. A. Alvarez. Using MEMS–based storage in disk arrays. *Conference on File and Storage Technologies* (San Francisco, 20, 31-02 April 2003), pages 89–101. USENIX Association, 2003.

[33] P. Vettiger, G. Cross, M. Despont, U. Drechsler, U. Dürig, B. Gotsmann, W. Häberle, M. A. Lantz, H. E. Rothuizen, R. Stutz, and G. K. Binnig. The "millipede": nanotechnology entering data storage. *IEEE Transactions on Nanotechnology*, **1**(1):39–55. IEEE, March 2002.

[34] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN, 16–20 May 1994), pages 241–251. ACM Press, 1994.

[35] H. Yu, D. Agrawal, and A. E. Abbadi. Tabular placement of relational data on MEMS-based storage devices. *International Conference on Very Large Databases* (Berlin, Germany, 09–12 September 2003), pages 680–693, 2003.

[36] H. Yu, D. Agrawal, and A. E. Abbadi. Towards optimal I/O scheduling for MEMS-based storage. *IEEE Symposium on Mass Storage Systems* (San Diego, CA, 07–10 April 2003), 2003.

[37] H. Yu, D. Agrawal, and A. E. Abbadi. *Declustering two-dimensional datasets over MEMS-based storage.* UCSB Department of Computer Science Technical Report 2003-27. September 2003.

[38] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 243–258. USENIX Association, 2000.

[39] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and scheduling an eager-writing disk array for a transaction processing workload. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 289–304. USENIX Association, 2002.

A Performance Comparison of NFS and iSCSI for IP-Networked Storage *

Peter Radkov, Li Yin‡, Pawan Goyal†, Prasenjit Sarkar†and Prashant Shenoy

Dept. of Computer Science
University of Massachusetts
Amherst MA 01003

†Storage Systems Research
IBM Almaden Research Center
San Jose CA 95120

‡Computer Science Division
University of California
Berkeley CA 94720

Abstract

IP-networked storage protocols such as NFS and iSCSI have become increasingly common in today's LAN environments. In this paper, we experimentally compare NFS and iSCSI performance for environments with no data sharing across machines. Our micro- and macro-benchmarking results on the Linux platform show that iSCSI and NFS are comparable for data-intensive workloads, while the former outperforms latter by a factor of two or more for meta-data intensive workloads. We identify aggressive meta-data caching and aggregation of meta-data updates in iSCSI to be the primary reasons for this performance difference and propose enhancements to NFS to overcome these limitations.

1 Introduction

With the advent of high-speed LAN technologies such as Gigabit Ethernet, IP-networked storage has become increasingly common in client-server environments. The availability of 10 Gb/s Ethernet in the near future is likely to further accelerate this trend. IP-networked storage is broadly defined to be any storage technology that permits access to remote data over IP. The traditional method for networking storage over IP is to simply employ a network file system such as NFS [11]. In this approach, the server makes a subset of its local namespace available to clients; clients access meta-data and files on the server using a RPC-based protocol (see Figure 1(a)).

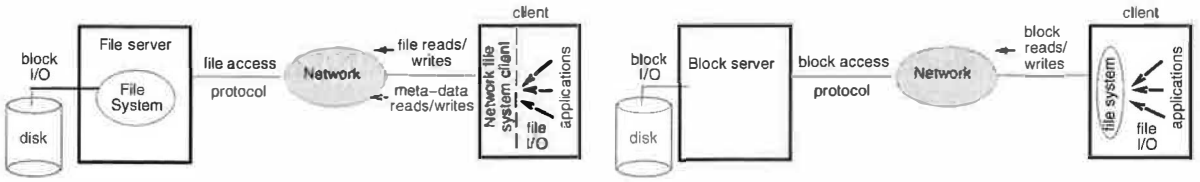
In contrast to this widely used approach, an alternate approach for accessing remote data is to use an IP-based storage area networking (SAN) protocol such as iSCSI [12]. In this approach, a remote disk exports a portion of its storage space to a client. The client handles the remote disk no differently than its local disks—it runs a local file system that reads and writes data blocks to the remote disk. Rather than accessing blocks from a local disk, the I/O operations are carried out over a network using a block access protocol (see Figure1(b)). In case of iSCSI, remote blocks are accessed by encapsulating SCSI commands into TCP/IP packets [12].

The two techniques for accessing remote data employ fundamentally different abstractions. Whereas a network file system accesses remote data at the granularity of files, SAN protocols access remote data at the granularity of disk blocks. We refer to these techniques as *file-access* and *block-access* protocols, respectively. Observe that, in the former approach, the file system resides at the server, whereas in the latter approach it resides at the client (see Figure 1). Consequently, the network I/O consists of file operations (file and meta-data reads and writes) for file-access protocols and block operations (block reads and writes) for block-access protocols.

Given these differences, it is not *a priori* clear which protocol type is better suited for IP-networked storage. In this paper, we take a first step towards addressing this question. We use NFS and iSCSI as specific instantiations of file- and block-access protocols and experimentally compare their performance. Our study specifically assumes an environment where a single client machine accesses a remote data store (i.e., there is no data sharing across machines), and we study the impact of the abstraction-level and caching on the performance of the two protocols.

Using a Linux-based storage system testbed, we carefully micro-benchmark three generations of the NFS protocols—NFS versions 2, 3 and 4, and iSCSI. We also measure application performance using a suite of data-intensive and meta-data intensive benchmarks such as PostMark, TPC-C and TPC-H on the two systems. We choose Linux as our experimental platform, since it is currently the only open-source platform to implement all three versions of NFS as well as the iSCSI protocol. The choice of Linux presents some challenges, since there are

*This research was supported in part by NSF grants CCR-9984030, EIA-0080119 and a gift from IBM Corporation.



(a) File-access Protocol (NFS) (b) Block-access Protocol (iSCSI)

Figure 1: An overview of file- and block-access protocols.

known performance issues with the Linux NFS implementation, especially for asynchronous writes and server CPU overhead. We perform detailed analysis to separate out the protocol behavior from the idiosyncrasies of the Linux implementations of NFS and iSCSI that we encounter during our experiments.

Broadly, our results show that, for environments in which storage is not shared across machines, iSCSI and NFS are comparable for data-intensive workloads, while the former outperforms the latter by a factor of two for meta-data intensive workloads. We identify aggressive meta-data caching and aggregation of meta-data updates in iSCSI as the primary reasons for this performance difference. We propose enhancements to NFS to extract these benefits of meta-data caching and update aggregation.

The rest of this paper is structured as follows. Section 2 provides a brief overview of NFS and iSCSI. Sections 3, 4, and 5 present our experimental comparison of NFS and iSCSI. Implications of our results are discussed in Section 6. Section 7 discusses our observed limitations of NFS and proposes an enhancement. Section 8 discusses related work, and we present our conclusions in Section 9.

2 Background: NFS and iSCSI

In this section, we present a brief overview of NFS and iSCSI and discuss their differences.

2.1 NFS Overview

There are three generations of the NFS protocol. In NFS version 2 (or simply "NFS v2"), the client and the server communicate via remote procedure calls (RPCs) over UDP. A key design feature of NFS version 2 is its stateless nature—the NFS server does not maintain any state about its clients, and consequently, no state information is lost if the server crashes.

The next version of NFS—NFS version 3—provides the following enhancements: (i) support for a variable length file handle of up to 64 bytes, instead of 32 byte files handles; (ii) eliminates the 8 KB limit on the maximum data transfer size; (iii) support for 64 bit offsets for file operations, up from 32 bits; (iv) reduces the number of fetch attribute calls by returning the file attributes on any call that modifies them; (v) supports asynchronous writes to improve performance; and (vi) adds support for TCP as a transport protocol in addition to UDP.

The latest version of NFS—NFS version 4—aims to improve the locking and performance for narrow data sharing applications. Some of the key features of NFS version 4 are as follows: (i) it integrates the suite of protocols (nfs, mountd, nlm, nsm) into one single protocol for ease of access across firewalls; (ii) it supports compound operations to coalesce multiple operations into one single message; (iii) it is *stateful* when compared to the previous incarnations of NFS — NFS v4 clients use OPEN and CLOSE calls for stateful interaction with the server; (iv) it introduces the concept of delegation to allow clients to aggressively cache file data; and (v) it mandates strong security using the GSS API.

2.2 iSCSI Overview

iSCSI is a block-level protocol that encapsulates SCSI commands into TCP/IP packets, and thereby leverages the investment in existing IP networks.

SCSI is a popular block transport command protocol that is used for high bandwidth transport of data between hosts and storage systems (e.g., disk, tape). Traditionally, SCSI commands have been transported over dedicated networks such as SCSI buses and Fiber Channel. With the emergence of Gigabit and 10 Gb/s Ethernet LANs, it is now feasible to transport SCSI commands over commodity networks and yet provide high throughput to bandwidth-intensive storage applications. To do so, iSCSI connects a SCSI initiator port on a host to a SCSI target port on a storage subsystem. For the sake of uniformity with NFS, we will refer to the initiator and the target as an iSCSI client and server, respectively.

Some of the salient features of iSCSI are as follows: (i) it uses the notion of a session between the client and the server to identify a communication stream between the two; (ii) it allows multiple connections to be multiplexed into a session; (iii) it supports advanced data integrity, authentication protocols as well as encryption (IPSEC)—these features are negotiated at session-startup time; and (iv) it supports advanced error recovery using explicit retransmission requests, markers and connection allegiance switching [12].

2.3 Differences Between NFS and iSCSI

NFS and iSCSI provide fundamentally different data sharing semantics. NFS is inherently suitable for data sharing, since it enable files to be shared among multiple client machines. In contrast, a block protocol such as iSCSI supports a single client for each volume on the block server. Consequently, iSCSI permits applications running on a single client machine to share remote data, but it is not directly suitable for sharing data *across* machines. It is possible, however, to employ iSCSI in shared multi-client environments by designing an appropriate distributed file system that runs on multiple clients and accesses data from block server.

The implications of caching are also different in the two scenarios. In NFS, the file system is located at the server and so is the file system cache (hits in this cache incur a network hop). NFS clients also employ a cache that can hold both data and meta-data. To ensure consistency across clients, NFS v2 and v3 require that client perform consistency checks with the server on cached data and meta-data. The validity of cached data at the client is implementation-dependent—in Linux, cached meta-data is treated as potentially stale after 3 seconds and cached data after 30 seconds. Thus, meta-data and data reads may trigger a message exchange (i.e., a consistency check) with the server even in the event of a cache hit. NFS v4 can avoid this message exchange for data reads if the server supports file delegation. From the perspective of writes, both data and meta-data writes in NFS v2 are synchronous. NFS v3 and v4 supports asynchronous data writes, but meta-data updates continue to be synchronous. Thus, depending on the version, NFS has different degrees of write-through caching.

In iSCSI, the caching policy is governed by the file system. Since the file system cache is located at the client, both data and meta-data reads benefit from any cached content. Data updates are asynchronous in most file systems. In modern file systems, meta-data updates are also asynchronous, since such systems use log-based journaling for faster recovery. In the ext3 file system, for instance, meta-data is written asynchronously at commit points. The asynchrony and frequency of these commit points is a trade-off between recovery and performance (ext3 uses a commit interval of 5 seconds). Thus, when used in conjunction with ext3, iSCSI supports a fully write-back cache for data and meta-data updates.

Observe that the benefits of asynchronous meta-data update in iSCSI come at the cost of lower reliability of data and meta-data persistence than in NFS. Due to synchronous meta-data updates in NFS, both data and meta-data updates persist across client failure. However, in iSCSI, meta-data updates as well as related data may be lost in case client fails prior to flushing the journal and data blocks to the iSCSI server.

3 Setup and Methodology

This section describes the storage testbed used for our experiments and then our experimental methodology.

3.1 System Setup

The storage testbed used in our experiments consists of a server and a client connected over an isolated Gigabit Ethernet LAN (see Figure 2). Our server is a dual processor machine with two 933 MHz Pentium-III processors, 256 KB L1 cache, 1 GB of main memory and an Intel 82540EM Gigabit Ethernet card. The server contains an Adaptec ServeRAID adapter card that is connected to a Dell PowerVault disk pack with fourteen SCSI disks; each disk is a 10,000 RPM Ultra-160 SCSI drive with 18 GB storage capacity. For the purpose of our experiments, we configure the storage subsystem as two identical RAID-5 arrays, each in a 4+p configuration (four data disks plus a parity disk). One array is used for our NFS experiments and the other for the iSCSI experiments. The client is a 1 GHz Pentium-III machine with 256KB L1 cache, 512 MB main memory, and an Intel 82540EM Gigabit Ethernet card.

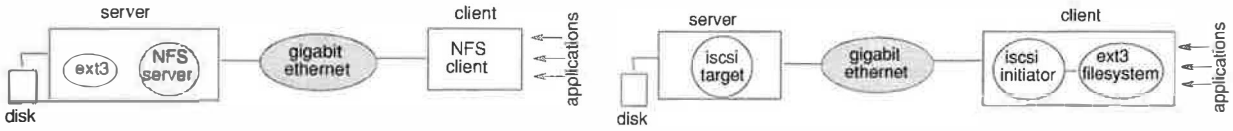
Both machines run RedHat Linux 9. We use version 2.4.20 of the Linux kernel on the client for all our experiments. For the server, we use version 2.4.20 as the default kernel, except for the iSCSI server which requires kernel version 2.4.2 and the NFS version 4 server which requires 2.4.18. We use the default Linux implementation of NFS versions 2 and 3 for our experiments. For NFS version 4, which is yet to be fully supported in vanilla Linux, we use the University of Michigan implementation (release 2 for Linux 2.4).

For iSCSI, we employ the open-source SourceForge Linux iSCSI implementation as the client (version 3.3.0.1) and a commercial implementation as the iSCSI server. While we found several high-quality open-source iSCSI client implementations, we were unable to find a stable open-source iSCSI server implementation that was compatible with our hardware setup; consequently, we chose a commercial server implementation.

The default file system used in our experiments is *ext3*. The file system resides at the client for iSCSI and at the server for NFS (see Figure 2). We use TCP as the default transport protocol for both NFS and iSCSI, except for NFS v2 where UDP is the transport protocol.

3.2 Experimental Methodology

We experimentally compare NFS versions 2, 3 and 4 with iSCSI using a combination of micro- and macro-benchmarks. The objective of our micro-benchmarking experiments is to measure the network message overhead of various file and directory operations in the two protocols, while our macro-benchmarks experimentally measure overall application performance.



(a) NFS setup (b) iSCSI setup

Figure 2: Experimental setup. The figures depict the setup used for our NFS and iSCSI experiments.

Our micro-benchmarks measure the network message overhead (number of network messages) for a variety of system calls that perform file and directory operations. We first measure the network message overhead assuming a cold cache at the client and the server and then repeat the experiment for a warm cache. By using a cold and warm cache, our experiments capture the worst and the average case, respectively, for the network message overhead. Since the network message overhead depends on the directory depth (path length), we also measure these overheads for varying directory depths. In case of file reads and writes, the network message overhead is dependent on (i) the I/O size, and (ii) the nature of the workload (i.e., random or sequential). Consequently, we measure the network message overhead for varying I/O sizes as well as sequential and random requests. We also study the impact of the network latency between the client and the server on the two systems.

We also measure application performance using several popular benchmarks: PostMark, TPC-C and TPC-H. PostMark is a file system benchmark that is meta-data intensive due its operation on a large number of small files. The TPC-C and TPC-H database benchmarks are data-intensive and represent online transaction processing and decision support application profiles.

We use a variety of tools to understand system behavior for our experiments. We use *Ethereal* to monitor network packets, the *Linux Trace toolkit* and *vmstat* to measure protocol processing times, and *nfsstat* to obtain nfs message statistics. We also instrument the Linux kernel to measure iSCSI network message overheads. Finally, we use logging in the VFS layer to trace the generation of network traffic for NFS. While we use these tools to obtain a detailed understanding of system behavior, reported performance results (for instance, for the various benchmarks) are without the various monitoring tools (to prevent the overhead of these tools from influencing performance results).

The next two sections provide a summary of our key experimental results. A more detailed presentation of the results can be found in [9].

4 Micro-benchmarking Experiments

This section compares the performance of various file and directory operations, focusing on protocol message counts as well as their sensitivity to file system parameters.

Table 1: File and directory-related system calls.

Directory operations	File operations
Directory creation (mkdir)	File create (creat)
Directory change (chdir)	File open (open)
Read directory contents (readdir)	Hard link to a file (link)
Directory delete (rmdir)	Truncate a file (truncate)
Symbolic link creation (symlink)	Change permissions (chmod)
Symbolic link read (readlink)	Change ownership (chown)
Symbolic link delete (unlink)	Query file permissions (access)
	Query file attributes (stat)
	Alter file access time (utime)

4.1 Overhead of System Calls

Our first experiment determines network message overheads for common file and directory operations at the granularity of system calls. We consider sixteen commonly-used system calls shown in Table 1 and measure their network message overheads using the *Ethereal* packet monitor. Note that this list does not include the *read* and *write* system calls, which are examined separately in Section 4.4.

For each system call, we first measure its network message overhead assuming a cold cache and repeat the experiment for a warm cache. We emulate a cold cache by unmounting and remounting the file system at the client and restarting the NFS server or the iSCSI server; this is done prior to each invocation of a system call. The warm cache is emulated by invoking the system call on a cold cache and then repeating the system call with similar (though not identical) parameters. For instance, to understand warm cache behavior, we create two directories in the same parent directory using `mkdir`, we open two files in the same directory using `open`, or we perform two different `chmod` operation on a file. In each case, the network message overhead of the second invocation is assumed to be the overhead in the presence of a warm cache.[1]

The directory structure can impact the network message overhead for a given operation. Consequently, we report overheads for a directory depth of zero and a directory depth of three. Section 4.3 reports additional results obtained by systematically varying the directory depth from 0 to 16.

[1] Depending on the exact cache contents, the warm cache network message overhead can be different for different caches. We carefully choose the system call parameters so as to emulate a "reasonable" warm cache. Moreover, we deliberately choose slightly different parameters across system call invocations; identical invocations will result in a hot cache (as opposed to a warm cache) and result in zero network message overhead for many operations.

Table 2: Network message overheads for a cold cache.

	Directory depth 0				Directory depth 3			
	V2	V3	V4	iSCSI	V2	V3	V4	iSCSI
mkdir	2	2	4	7	5	5	10	13
chdir	1	1	3	2	4	4	9	8
readdir	2	2	4	6	5	5	10	12
symlink	3	2	4	6	6	5	10	12
readlink	2	2	3	5	5	5	9	10
unlink	2	2	4	6	5	5	10	11
rmdir	2	2	4	8	5	5	10	14
creat	3	3	10	7	6	6	16	13
open	2	2	7	3	5	5	13	9
link	4	4	7	6	10	9	16	12
rename	4	3	7	6	10	10	16	12
trunc	3	3	8	6	6	6	14	12
chmod	3	3	5	6	6	6	11	12
chown	3	3	5	6	6	6	11	11
access	2	2	5	3	5	5	11	9
stat	3	3	5	3	6	6	11	9
utime	2	2	4	6	5	5	10	12

Table 2 depicts the number of messages exchanged between the client and server for NFS versions 2, 3, 4 and iSCSI assuming a cold cache.

We make three important observations from the table. First, on an average, iSCSI incurs a higher network message overhead than NFS. This is because a single message is sufficient to invoke a file system operation on a path name in case of NFS. In contrast, the path name must be completely resolved in case of iSCSI before the operation can proceed; this results in additional message exchanges. Second, the network message overhead increases as we increase the directory depth. For NFS, this is due to the additional access checks on the pathname. In case of iSCSI, the file system fetches the directory inode and the directory contents at each level in the path name. Since directories and their inodes may be resident on different disk blocks, this triggers additional block reads. Third, NFS version 4 has a higher network message overhead when compared to NFS versions 2 and 3, which have a comparable overhead. The higher overhead in NFS version 4 is due to access checks performed by the client via the *access* RPC call.[2]

We make one additional observation that is not directly reflected in Table 2. The average message size in iSCSI can be higher than that of NFS. Since iSCSI is a block access protocol, the granularity of reads and writes in iSCSI is a disk block, whereas RPCs allow NFS to read or write smaller chunks of data. While reading entire blocks may seem wasteful, a side-effect of this policy is that iSCSI benefits from aggressive caching. For instance, reading an entire disk block of inodes enable applications with meta-data locality to benefit in iSCSI. In the absence of meta-data or data locality, however, reading entire disk blocks may hurt performance.

While the message size can be an important contributor to the network message overhead analysis of the two protocols, our observations in the macro-benchmark analysis indicated that the number of messages exchanged was the dominant factor in the network message overhead. Consequently, we focus on the number of messages exchanged as the key factor in network message overhead in the rest of the analysis.

Table 3: Network message overheads for a warm cache.

	Directory depth 0				Directory depth 3			
	v2	v3	v4	iSCSI	v2	v3	v4	iSCSI
mkdir	2	2	2	2	4	4	3	2
chdir	1	1	0	0	3	3	2	0
readdir	1	1	0	2	3	3	3	2
symlink	3	2	2	2	5	4	4	2
readlink	1	2	0	2	3	3	3	2
unlink	2	2	2	2	5	4	3	2
rmdir	2	2	2	2	4	4	3	2
open	3	2	6	2	5	5	9	2
creat	4	3	2	2	6	4	6	2
open	1	1	4	0	4	4	6	0
rename	4	3	2	2	6	6	6	2
trunc	2	2	4	2	5	5	7	2
chmod	2	2	2	2	4	5	5	2
chown	2	2	2	2	4	5	5	2
access	1	1	1	2	4	4	3	0
stat	2	2	2	2	5	5	5	0
utime	1	1	1	2	4	4	4	2

Table 3 depicts the number of messages exchanged between the client and the server for warm cache operations. Whereas iSCSI incurred a higher network message overhead than NFS in the presence of a cold cache, it incurs a comparable or lower network message overhead than NFS in the presence of a warm cache. Further, the network message overhead is identical for directory depths of zero and three for iSCSI, whereas it increases with directory depth for NFS. Last, both iSCSI and NFS benefit from a warm cache and the overheads for each operation are smaller than those for a cold cache. The better performance of iSCSI can be attributed to aggressive meta-data caching performed by the file system; since the file system is resident at the client, many requests can be serviced directly from the client cache. This is true even for long path names, since all directories in the path may be cached from a prior operation. NFS is unable to extract these benefits despite using a client-side cache, since NFS v2 and v3 need to perform consistency checks on cached entries, which triggers message exchanges with the server. Further, meta-data update operations are necessarily synchronous in NFS, while they can be asynchronous in iSCSI. This asynchronous nature enables applications to update a dirty cache block multiple times prior to a flush, thereby amortizing multiple meta-data updates into a single network block write.

[2]The *access* RPC call was first introduced in NFS V3. Our Ethereal logs did not reveal its use in the Linux NFS v3 implementation, other than for root access checks. However, the NFS v4 client uses it extensively to perform additional access checks on directories and thereby incurs a higher network message overhead.

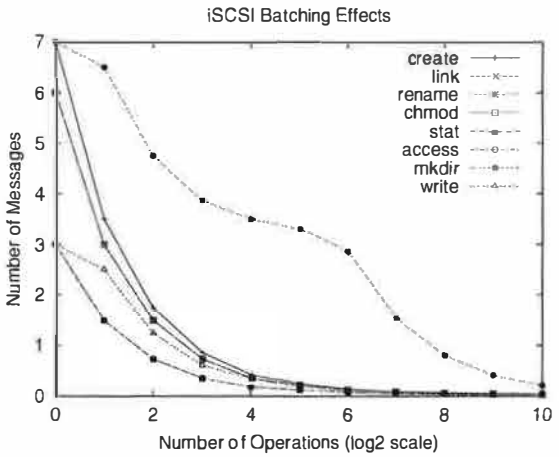


Figure 3: Benefit of meta-data update aggregation and caching in iSCSI. The figure shows the amortized network message overhead per operation for varying batch sizes. The batch size is shown on a logarithmic scale.

4.2 Impact of Meta-data Caching and Update Aggregation

Our micro-benchmark experiments revealed two important characteristics of modern local file systems — aggressive meta-data caching, which benefits meta-data reads, and update aggregation, which benefits meta-data writes. Recall that, update aggregation enables multiple writes to the same dirty block to be "batched" into a single asynchronous network write. We explore this behavior further by quantifying the benefits of update aggregation and caching in iSCSI.

We choose eight common operations that read and update meta-data, namely `creat`, `link`, `rename`, `chmod`, `stat`, `access`, `write` and `mkdir`. For each operation, we issue a batch of N consecutive calls of that operation and measure the network message overhead of the entire batch. We vary N from 1 to 1024 (e.g., issue 1 mkdir, 2 mkdirs, 4 mkdirs and so on, while starting with a cold cache prior to each batch). Figure 3 plots the amortized network message overhead per operation for varying batch sizes. As shown, the amortized overhead reduces significantly with increasing batch sizes, which demonstrates that update aggregation can indeed significantly reduce the number of network writes. Note that some of the reduction in overhead can be attributed to meta-data caching in iSCSI. Since the cache is warm after the first operation in a batch, subsequent operations do not yield additional caching benefits — any further reduction in overhead is solely due to update aggregation. In general, our experiment demonstrates applications that exhibit meta-data locality can benefit significantly from update aggregation.

4.3 Impact of Directory Depth

Our micro-benchmarking experiments gave a preliminary indication of the sensitivity of the network message overhead to the depth of the directory where the file operation was performed. In this section, we examine this sensitivity in detail by systematically varying the directory depth.

For each operation, we vary the directory depth from 0 to 16 and measure the network message overhead in NFS and iSCSI for the cold and warm cache. A directory depth of i implies that the operation is executed in $mnt\_point : /dir1/.../diri$. Figure 4 lists the observed overhead for three different operations.

In the case of cold cache, iSCSI needs two extra messages for each increase in directory depth due to the need to access the directory inode as well as the directory contents. In contrast, NFS v2 and v3 need only one extra message for each increase in directory depth, since only one message is needed to access directory contents — the directory inode lookup is done by the server. As indicated earlier, NFS v4 performs an extra access check on each level of the directory via the *access* call. Due to this extra message, its overhead matches that of iSCSI and increases in tandem.[3] Consequently, as the directory depth is increased, the iSCSI overhead increases faster than NFS for the cold cache.

In contrast, a warm cache results in a constant number of messages independent of directory depth due to meta-data caching at the client for both NFS and iSCSI. The observed messages are solely due to the need to update meta-data at the server.

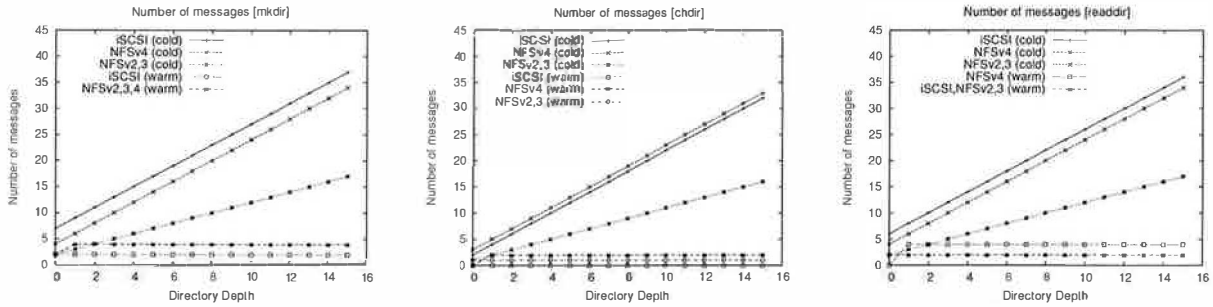
4.4 Impact of Read and Write Operations

Our experiments thus far have focused on meta-data operations. In this section, we study the efficiency of data operations in NFS and iSCSI. We consider the *read* and *write* system calls and measure their network message overheads in the presence of a cold and a warm cache.

To measure the read overhead, we issue reads of varying sizes — 128 bytes to 64 KB — and measure the resulting network message overheads in the two systems. For the warm cache, we first read the entire file into the cache and then issue sequential reads of increasing sizes. The write overhead is measured similarly for varying write sizes. The cold cache is emulated by emptying the client and server caches prior to the operation. Writes are however not measured in warm cache mode — we use macro-benchmarks to quantify warm cache effects.

Figure 5 plots our results. We make the following observations from our results. For read operations, iSCSI requires one or two extra messages over NFS to read

[3]The extra overhead of *access* is probably an artifact of the implementation. It is well-known that the Linux NFS implementation does not correctly implement the *access* call due to inadequate caching support at the client [7]. This idiosyncrasy of Linux is the likely cause of the extra overhead in NFS v4.



(a) `mkdir` (b) `chdir` (c) `readdir`

Figure 4: Effect of the directory depth on the network message overhead.

or update uncached file meta-data (e.g., inode blocks). While NFS incurs a smaller overhead for small cold reads, the read overhead exceeds that of iSCSI beyond 8KB requests. For NFS v2, this is due to the maximum data transfer limit of 8KB imposed by the protocol specification. Multiple data transfers are needed when the read request size exceeds this limit. Although NFS v3 eliminates this restriction, it appears that the Linux NFS v3 implementation does not take advantage of this flexibility and uses the same transfer limit as NFS v2. Consequently, the cold read overhead of NFS v3 also increases beyond that of iSCSI for large reads. In contrast, the NFS v4 implementation uses larger data transfers and incurs fewer messages. In case of the warm cache, since the file contents are already cached at the client, the incurred overhead in NFS is solely due to the consistency checks performed by the client. The observed overhead for iSCSI is due to the need to update the access time in the inode.

Similar observations are true for write requests (see Figure 5(c)). Initially, the overhead of iSCSI is higher primarily due to the need to access uncached meta-data blocks. For NFS, all meta-data lookups take place at the server and the network messages are dominated by data transfers. The network message overhead for NFS v2 increases once the write request size exceeds the maximum data transfer limit; the overhead remains unchanged for NFS v3 and 4.

4.5 Impact of Sequential and Random I/O

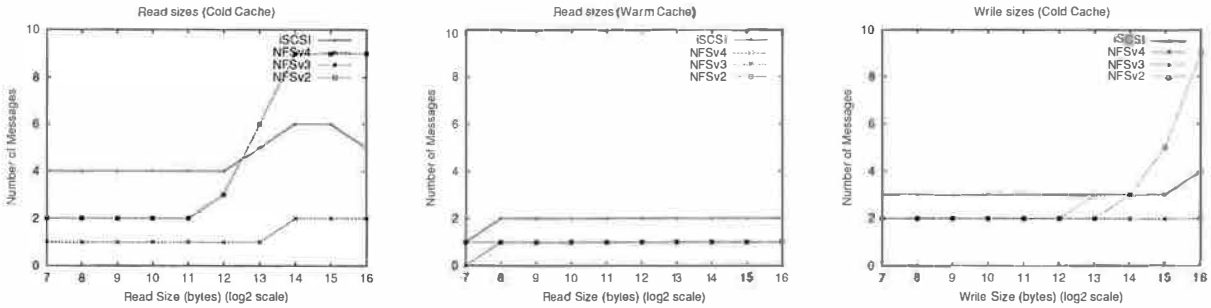
Two key factors impact the network message overheads of data operations—the size of read and write requests and the access characteristics of the requests (sequential or random). The previous section studied the impact of request sizes on the network message overhead. In this section, we study the effect of sequential and random access patterns on network message overheads.

To measure the impact of reads, we create a 128MB file. We then empty the cache and read the file sequentially in 4KB chunks. For random reads, we create a random permutation of the 32K blocks in the file and read the blocks in that order. We perform this experiment first for NFS v3 and then for iSCSI. Table 4 depicts the completion times, network message overheads and bytes transferred in the two systems. As can be seen, for sequential reads, both NFS and iSCSI yield comparable performance. For random reads, NFS is slightly worse (by about 15%). The network message overheads and the bytes transfered are also comparable for iSCSI and NFS.

Next, we repeat the above experiment for writes. We create an empty file and write 4KB data chunks sequentially to a file until the file size grows to 128MB. For random writes, we generate a random permutation of the 32K blocks in the file and write these blocks to newly created file in that order. Table 4 depicts our results. Unlike reads, where NFS and iSCSI are comparable, we find that iSCSI is significantly faster than NFS for both sequential and random writes. The lower completion time of iSCSI is due to the asynchronous writes in the ext3 file system. Since NFS version 3 also supports asynchronous writes, we expected the NFS performance to be similar to iSCSI. However, it appears that the Linux NFS v3 implementation can not take full advantage of asynchronous writes, since it specifies a limit on the number of pending writes in the cache. Once this limit is exceeded, the write-back caches degenerates to a write-through cache and application writes see a pseudo-synchronous behavior. Consequently, the NFS write performance is significantly worse than iSCSI. Note also, while the byte overhead is comparable in the two systems, the number of messages in iSCSI is significantly smaller than NFS. This is because iSCSI appears to issue very large write requests to the server (mean request size is 128KB as opposed to 4.7KB in NFS).

4.6 Impact of Network Latency

Our experiments thus far have assumed a lightly loaded Gigabit Ethernet LAN. The observed round trip times on our LAN is very small (<1ms). In practice, the latency between the client and the server can vary from a few milliseconds to tens of milliseconds depending on the



(a) Cold reads (b) Warm reads (c) Cold Writes

Figure 5: Network message overheads of read and write operations of varying sizes.

Table 4: Sequential and Random reads and writes: completion times, number of messages and bytes transferred for reading and writing a 128MB file.

	Performance		Messages		Bytes	
	NFS v3	iSCSI	NFS v3	iSCSI	NFS v3	iSCSI
Sequential reads	35s	35s	33,362	32,790	153MB	148MB
Random reads	64s	55s	32,860	32,827	153MB	148MB
Sequential writes	17s	2s	32,990	1135	151MB	143MB
Random writes	21s	5s	33,015	1150	151MB	143MB

distance between the client and the server. Consequently, in this section, we vary the network latency between the two machines and study its impact on performance.

We use the NISTNet package to introduce a latency between the client and the server. NISTNet introduces a pre-configured delay for each outgoing and incoming packet so as to simulate wide-area conditions. We vary the round-trip network latency from 10ms to 90ms and study its impact on the sequential and random reads and writes. The experimental setup is identical to that outlined in the previous section. Figure 6 plots the completion times for reading and writing a 128 MB file for NFS and iSCSI. As shown in Figure 6(a), the completion time increases with the network latency for both systems. However, the increase is greater in NFS than in iSCSI—the two systems are comparable at low latencies ($\leq$ 10ms) and the NFS performance degrades faster than iSCSI for higher latencies. Even though NFS v3 runs over TCP, an *Ethereal* trace reveals an increasing number of RPC retransmissions at higher latencies. The Linux NFS client appears to time-out more frequently at higher latencies and reissues the RPC request, even though the data is in transit, which in turn dregrades performance. An implementation of NFS that exploits the error recovery at the TCP layer will not have this drawback.

In case of writes, the iSCSI completion times are not affected by the network latency due to their asynchronous nature. The NFS performance is impacted by the pseudo-synchronous nature of writes in the Linux NFS implementation (see Section 4.5) and increases with the latency.

5 Macro-benchmarking Experiments

This section compares the overall application level performance for NFS v3 and iSCSI.

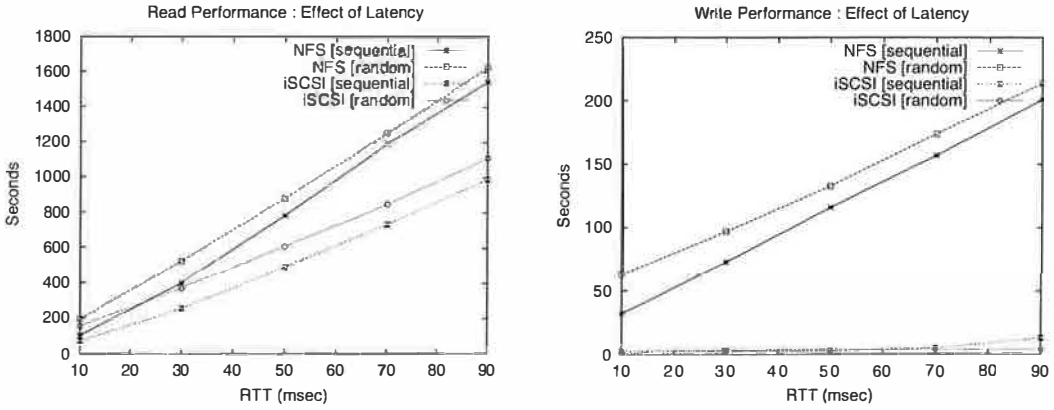
5.1 PostMark Results

PostMark is a benchmark that demonstrates system performance for short-lived small files seen typically in Internet applications such as electronic mail, netnews and web-based commerce. The benchmark creates an initial pool of random text files of varying size. Once the pool has been created, the benchmark performs two types of transactions on the pool: (i) create or delete a file; (ii) read from or append to a file. The incidence of each transaction and its subtype are chosen randomly to eliminate the effect of caching and read-ahead.

Our experiments use a equal predisposition to each type of transaction as well as each subtype within a transaction. We performed 100,000 transactions on a pool of files whose size was varied from 1,000 to 25,000 in multiples of 5.

Table 5 depicts our results. As shown in the table, iSCSI generally outperforms NFS v3 due to the meta-data intensive nature of this benchmark. An analysis of the NFS v3 protocol messages exchanged between the server and the client shows that 65% of the messages are meta-data related. Meta-data update aggregation as well as aggressive meta-data caching in iSCSI enables it to have a significantly lower message count than NFS.

As the pool of files is increased, we noted that the



(a) Reads (b) Writes

Figure 6: Impact of network latency on read and write performance.

Table 5: PostMark Results. Completion times and message counts are reported for 100,000 operations on 1,000, 5,000 and 25,000 files.

	Completion time (s)		Messages	
Files	NFS v3	iSCSI	NFS v3	iSCSI
1,000	146	12	371,963	101
5,000	201	35	451,415	276
25,000	516	208	639,128	66,965

benefits of meta-data caching and meta-data update aggregation starts to diminish due to the random nature of the transaction selection. As can be seen in Table 5, the number of messages relative to the file pool size increases faster in iSCSI than that in NFS v3. Consequently, the performance difference between the two decreases. However, as a side effect, the benchmark also reduces the effectiveness of meta-data caching on the NFS server, leading to higher server CPU utilization (see Section 5.4).

5.2 TPC-C and TPC-H Results

TPC-C is an On-Line Transaction Processing (OLTP) benchmark that leads to small 4 KB random I/Os. Two-thirds of the I/Os are reads. We set up TPC-C with 300 warehouses and 30 clients. We use IBM's DB2 database for Linux (version 8.1 Enterprise Edition). The metric for evaluating TPC-C performance is the number of transactions completed per minute (tpmC).

Table 6 shows the TPC-C performance and the network message overhead for NFS and iSCSI. Since these are results from an unaudited run, we withhold the actual results and instead report normalized throughput for the two systems.[4] As shown in the table, there is a marginal

[4]The Transaction Processing Council does not allow unaudited results to be reported.

Table 6: TPC-C Results. Reported throughput (tpmC) is normalized by a factor α equivalent to the throughput obtained with NFS v3.

Peak Throughput (TpmC)		Messages	
NFS v3	iSCSI	NFS v3	iSCSI
α	1.08 $*\alpha$	517,219	530,745

difference between NFS v3 and iSCSI. This is not surprising since TPC-C is primarily data-intensive and as shown in earlier experiments, iSCSI and NFS are comparable for data-intensive workloads. An analysis of the message count shows that the vast majority of the NFS v3 protocol traffic (99%) is either a data read or a data write. The two systems are comparable for read operations. Since data writes are 4KB each and less-intensive than in other benchmarks, NFS is able to benefit from asynchronous write support and is comparable to iSCSI.

The TPC-H benchmark emulates a decision support systems that examines large volumes of data, executes queries with a high degree of complexity, and gives answers to critical business questions. Our TPC-H experiments use a database scale factor of 1 (implying a 1 GB database). The page size and the extent size for the database were chosen to be 4 KB and 32 KB, respectively. We run the benchmark for iSCSI and NFS and report the observed throughout and network message overheads in Table 7. Again, we report normalized throughputs since our results are unaudited. The reported throughput for TPC-H is the number of queries per hour for a given database size (QphH@1GB in our case).

We find the performance of NFS and iSCSI is comparable for TPC-H. Since the benchmark is dominated by large read requests—an analysis of the traffic shows that the vast majority of the messages are data reads—this result is consistent with prior experiments where iSCSI and NFS were shown to have comparable performance

Table 7: TPC-H Results. Reported throughput (QphH@1GB) is normalized by a factor β equivalent to the throughput obtained in NFS v3.

Throughput (QphH@1GB)		Messages	
NFS v3	iSCSI	NFS v3	iSCSI
β	1.07 $*\beta$	261,769	62,686

Table 8: Completion times for other benchmarks.

Benchmark	NFS v3	iSCSI
`tar -xzf`	60s	5s
`ls -lR > /dev/null`	12s	6s
`kernel compile`	222s	193s
`rm -rf`	40s	22s

for read-intensive workloads.

Workloads dominated by large sequential reads can also signify the maximum application throughput that be sustained by a protocol. The experiments indicate no perceptible difference in this particular edge-condition case.

5.3 Other Benchmarks

We also used several simple macro-benchmarks to characterize the performance of iSCSI and NFS. These benchmarks include extracting the Linux kernel source tree from a compressed archive (tar xfz), listing the contents (ls -lR), compiling the source tree (make) and finally removing the entire source tree (rm -rf). The first, second and fourth benchmarks are met-data intensive and amenable to meta-data caching as well as meta-data update aggregation. Consequently, in these benchmarks, iSCSI performs better than NFS v3. The third benchmark, which involves compiling the Linux kernel, is CPU-intensive, and consequently there is parity between iSCSI and NFS v3. The marginal difference between the two can be attributed to the impact of the iSCSI protocol's reduced processing length on the single-threaded compiling process.

5.4 CPU utilization

A key performance attribute of a protocol is its scalability with respect to the number of clients that can be supported by the server. If the network paths or I/O channels are not the bottleneck, the scalability is determined by the server CPU utilization for a particular benchmark.

Table 9 depicts the 99^{th} percentile of the server CPU utilization reported every 2 seconds by `vmstat` for the various benchmarks. The table shows that, the server utilization for iSCSI is lower than that of NFS. The server utilization is governed by the processing path and the amount of processing for each request. The lower utilization of iSCSI can be attributed to the smaller processing path seen by iSCSI requests. In case of iSCSI, a block read or write request at the server traverses through the network layer, the SCSI server layer, and the low-level block device driver. In case of NFS, an RPC call received by the server traverses through the network layer, the NFS server layer, the VFS layer, the local file system, the block layer, and the low-level block device driver. Our measurements indicate that the server processing path for NFS requests is twice that of iSCSI requests. This is confirmed by the server CPU utilization measurements for data intensive TPC-C and TPC-H benchmarks. In these benchmarks, the server CPU utilization in for NFS is twice that of iSCSI.

Table 9: Server CPU utilization for various benchmarks. The 99^{th} percentile of the CPU utilization at the server is reported for each benchmark.

	NFS v3	iSCSI
PostMark	77%	13%
TPC-C	13%	7%
TPC-H	20%	11%

The difference is exacerbated for meta-data intensive workloads. A NFS request that triggers a meta-data lookup at the server can greatly increase the processing path—meta-data reads require multiple traversals of the VFS layer, the file system, the block layer and the block device driver. The number of traversals depends on the degree of meta-data caching in the NFS server. The increased processing path explains the large disparity in the observed CPU utilizations for PostMark. The PostMark benchmark tends to defeat the meta-data caching on the NFS server because of the random nature of transaction selection. This causes the server CPU utilization to increase significantly since multiple block reads may be needed to satisfy a single NFS data read.

While the iSCSI protocol demonstrates a better profile in server CPU utilization statistics, it is worthwhile to investigate the effect of these two protocols on client CPU utilization. If the client CPU utilization of one protocol has a better profile than that of the other protocol, then the first protocol will be able to scale to a larger number of servers per client.

Table 10 depicts the 99^{th} percentile of the client CPU utilization reported every 2 seconds by `vmstat` for the various benchmarks. For the data-intensive TPC-C and TPC-H benchmarks, the clients are CPU saturated for both the NFS and iSCSI protocols and thus there is no difference in the client CPU utilizations for these macro-benchmarks. However, for the meta-data intensive PostMark benchmark, the NFS client CPU utilization is an order of magnitude lower than that of iSCSI. This is not surprising because the bulk of the meta-data processing

Table 10: Client CPU utilization for various benchmarks. The 99^{th} percentile of the CPU utilization at the server is reported for each benchmark.

	NFS v3	iSCSI
PostMark	2%	25%
TPC-C	100%	100%
TPC-H	100%	100%

is done at the server in the case of NFS while the reverse is true in the case of the iSCSI protocol.

6 Discussion of Results

This section summarizes our results and discuss their implications for IP-networked storage in environments where storage in not shared across multiple machines.

6.1 Data-intensive applications

Overall, we find that iSCSI and NFS yield comparable performance for data-intensive applications, with a few caveats for write-intensive or mixed workloads.

In particular, we find that any application that generates predominantly read-oriented network traffic will see comparable performance in iSCSI and NFS v3. Since NFS v4 does not make significant changes to those portions of the protocol that deal with data transfers, we do not expect this situation to change in the future. Furthermore, the introduction of hardware protocol acceleration is likely to improve the data transfer part of both iSCSI and NFS in comparable ways.

In principle, we expect iSCSI and NFS to yield comparable performance for write-intensive workloads as well. However, due to the idiosyncrasies of the Linux NFS implementation, we find that iSCSI significantly outperforms NFS v3 for such workloads. We believe this is primarily due to the limit on the number of pending asynchronous writes at the NFS client. We find that this limit is quickly reached for very write-intensive workloads, causing the write-back cache at the NFS client to degenerate into a write-through cache. The resulting pseudo-synchronous write behavior causes a substantial performance degradation (by up to an order of magnitude) in NFS. We speculate that an increase in the pending writes limit and optimizations such as spatial write aggregation in NFS will eliminate this performance gap.

Although the two protocols yield comparable application performance, we find that they result in different server CPU utilizations. In particular, we find that the server utilization is twice as high in NFS than in iSCSI. We attribute this increase primarily due to the increased processing path in NFS when compared to iSCSI. An implication of the lower utilization in iSCSI is that the server is more scalable (i.e., it can service twice as many clients with the caveat that there is no sharing between client machines). It is worth noting that NFS appliances use specialized techniques such as cross-layer optimizations and hardware acceleration support to reduce server CPU utilizations by an order of magnitude – the relative effect of these techniques on NFS and iSCSI servers is a matter of future research.

6.2 Meta-data intensive applications

NFS and iSCSI show their greatest differences in their handling of meta-data intensive applications. Overall, we find that iSCSI outperforms NFS for meta-data intensive workloads—workloads where the network traffic is dominated by meta-data accesses.

The better performance of iSCSI can be attributed to two factors. First, NFS requires clients to update meta-data synchronously to the server. In contrast, iSCSI, when used in conjunction with modern file systems, updates meta-data asynchronously. An additional benefit of asynchronous meta-data updates is that it enables update aggregation—multiple meta-data updates to the same cached block are aggregated into a single network write, yielding significant savings. Such optimizations are not possible in NFS v2 or v3 due to their synchronous meta-data update requirement.

Second, iSCSI also benefits from aggressive meta-data caching by the file system. Since iSCSI reads are in granularity of disk blocks, the file system reads and caches entire blocks containing meta-data; applications with meta-data locality benefit from such caching. Although the NFS client can also cache meta-data, NFS clients need to perform periodic consistency checks with the server to provide weak consistency guarantees across client machines that share the same NFS namespace. Since the concept of sharing does not exist in the SCSI architectural model, the iSCSI protocol also does not pay the overhead of such a consistency protocol.

6.3 Applicability to Other File Protocols

An interesting question is the applicability of our results to other protocols such as NFS v4, DAFS, and SMB.

The SMB protocol is similar to NFS v4 in that both provide support for strong consistency. Consistency is ensured in SMB by the use of opportunistic locks or oplocks which allow clients to have exclusive access over a file object. The DAFS protocol specification is based on NFS v4 with additional extensions for hardware-accelerated performance, locking and failover. These extensions do not affect the basic protocol exchanges that we observed in our performance analysis.

NFS v4, DAFS and SMB do not allow a client to update meta-data asynchronously. NFS v4 and DAFS allow the use of compound RPCs to aggregate related meta-data requests and reduce network traffic. This can improve performance in meta-data intensive benchmarks

such as PostMark. However, it is not possible to speculate on the actual performance benefits, since it depends on the degree of compounding.

6.4 Implications

Extrapolating from our NFS and iSCSI results, it appears that block- and file-access protocols are comparable on data-intensive benchmarks and the former outperforms the latter on the meta-data intensive benchmarks. From the perspective of performance for IP-networked storage in an unshared environment, this result favors a block-access protocol over a file-access protocol. However, the choice between the two protocols may be governed by other significant considerations not addressed by this work such as ease of administration, availability of mature products, cost, etc.

Observe that the meta-data performance of the NFS protocol suffers primarily because it was designed for sharing of files across clients. Thus, when used in an environment where files are not shared, the protocol pays the penalty of features designed to enable sharing. There are two possible ways to address this limitation: (1) Design a file-access protocol for an unshared environments; and (2) Extend the NFS protocol so that while it provides sharing of files when desired, it does not pay the penalty of "sharing" when files are not shared. Since sharing of files is desirable, we propose enhancements to NFS in Section 7 that achieve the latter goal.

7 Potential Enhancements for NFS

Our previous experiments identified three factors that affect NFS performance for meta-data-intensive applications: (i) consistency check related messages (ii) synchronous meta-data update messages and (iii) non-aggregated meta-data updates. This section explores enhancements that eliminate these overheads.

The consistency check related messages can be eliminated by using a strongly-consistent read-only name and attribute cache as proposed in [13]. In such a cache, meta-data read requests are served out of the local cache. However, all update requests are forwarded to the server. On an update of an object, the server invalidates the caches of all clients that have that object cached.

The meta-data updates can be made asynchronously in an aggregated fashion by enhancing NFS to support *directory delegation*. In directory delegation a NFS client holds a lease on meta-data and can update and read the cached copy without server interaction. Since NFS v4 only supports file delegation, directory delegation would be an extension to the NFS v4 protocol specification. Observe that directory delegation allows a client to asynchronously update meta-data in an aggregated fashion. This in turn would allow NFS clients to have comparable performance with respect to iSCSI clients even for meta-data update intensive benchmarks. Directory delegation can be implemented using leases and callbacks [4].

The effectiveness of strongly-consistent read-only meta-data cache as well as directory delegation depends on the amount of meta-data sharing across client machines. Hence, we determine the characteristics of meta-data sharing in NFS by analyzing two real-world NFS workload traces from Harvard University [2]. We randomly choose one day (09/20/2001) trace from the EECS traces (which represents a research, software development, and course-based workload) and the home02 trace from the Campus traces (which represents a email and web workload). Roughly 40,000 file system objects were accessed for the EECS traces and about 100,000 file system objects were visited for the Campus traces.

Figure 7 demonstrates that the read sharing of directories is much higher than write sharing in the EECS trace. In Campus trace, we find that although the read-sharing is higher at smaller time-scales, it is less than the read-write sharing at larger time-scales. However, in both the traces, a relatively small percentage of directories are both read and written by multiple clients. For example, at time-scale of 300 seconds only 4% and 3.5% percentage of directories are read-write shared in EECS and Campus traces, respectively. This suggests that cache invalidation rate in strongly consistent meta-data read cache and contention for leases in directory delegation should not be significant, and it should be possible to implement both techniques with low overhead.

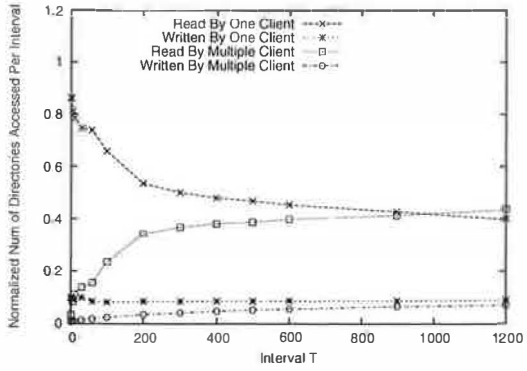
We evaluated the utility of strongly-consistent read-only meta-data caching using simulations. Our simulation results demonstrated that a directory cache size of 5 leads to more than 80% reduction in meta-data messages. Furthermore, the number of messages for cache invalidation is fairly low. The callback ratio, defined as ratio of cache-invalidation messages and number of meta-data messages , is less than 0.4% for a directory cache size of 5 for the EECS and campus traces.

The above preliminary results indicate that implementing a strongly-consistent read-only meta-data cache and directory delegation is feasible and would enable a NFS v4 client with these enhancements to have comparable performance with respect to an iSCSI client even for meta-data intensive benchmarks. A detailed design of these enhancements and their performance is beyond the scope of this paper and is the subject of future research.

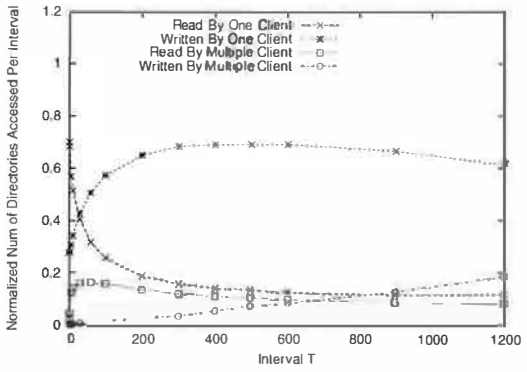
8 Related Work

Numerous studies have focused on the performance and cache consistency of network file-access protocols [4, 8, 11, 13]. In particular, the benefits of meta-data caching in a distributed file system for a decade old workload were evaluated in [13].

The VISA architecture was notable for using the con-



(a) EECS Trace



(b) Campus Trace

Figure 7: Sharing Characteristics of Directories

cept of SCSI over IP[6]. Around the same time, a parallel effort from CMU also proposed two innovative architectures for exposing block storage devices over a network for scalability and performance [3].

Several studies have focused in the performance of the iSCSI protocol from the perspective of on data path overheads and latency[1, 5, 12]. With the exception of [5], which compares iSCSI to SMB, most of these efforts focus solely on iSCSI performance. Our focus is different in that we examine the suitability of block- and file-level abstractions for designing IP-networked storage. Consequently, we compare iSCSI and NFS along several dimensions such as protocol interactions, network latency and sensitivity to different application workloads. A recent white paper [14] compares a commercial iSCSI target implementation and NFS using meta-data intensive benchmarks. While their conclusions are similar to ours for these workloads, our study is broader in its scope and more detailed.

A comparison of block- and file-access protocols was first carried out in the late eighties [10]. This study predated both NFS and iSCSI and used analytical modeling to compare the two protocols for DEC's VAX systems. Their models correctly predicted higher server CPU utilizations for file access protocols as well as the need for data and meta-data caching in the client for both protocols. Our experimental study complements and corroborates these analytical results for modern storage systems.

9 Concluding Remarks

In this paper, we use NFS and iSCSI as specific instantiations of file- and block-access protocols and experimentally compare their performance in environments where storage is not shared across client machines. Our results demonstrate that the two are comparable for data-intensive workloads, while the former outperforms the latter by a factor of 2 or more for meta-data intensive workloads. We identify aggressive meta-data caching and update aggregation allowed by iSCSI to be the primary reasons for this performance difference. We propose enhancements to NFS to improve its meta-data performance and present preliminary results that show its effectiveness. As part of future work, we plan to implement this enhancement in NFS v4 and study its performance for real application workloads.

Acknowledgments

We thank the anonymous reviewers and our shepherd Greg Ganger for their comments.

References

[1] S Aiken, D. Grunwald, A. Pleszkun, and J. Willeke. A Performance Analysis of the iSCSI Protocol. In *Proceedings of the 20th IEEE Symposium on Mass Storage Systems, San Diego, CA*, April 2003.

[2] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of USENIX FAST'03*, San Francisco, CA, March 2003.

[3] G A. Gibson et. al. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), San Jose, CA*, pages 92–103, Oct 1998.

[4] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[5] Y. Lu and D. Du. Performance Study of iSCSI-Based Storage Subsystems. *IEEE Communications Magazine*, August 2003.

[6] R. Van Meter, G. Finn, and S. Hotz. VISA: Netstation's Virtual Internet SCSI Adapter. In *Proceedings of ASPLOS-VIII, San Jose, CA*, pages 71–80, 1998.

[7] T. Myklebust. Status of the Linux NFS Client. Presentation at Sun Microsystems Connectathon 2002, http://www.connectathon.org/talks02, 2002.

[8] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. In *Proceedings of the Summer 1994 USENIX Conference*, June 1994.

[9] P. Radkov, Y. Li, P. Goyal, P. Sarkar, and P. Shenoy. An Experimental Comparison of File- and Block-Access Protocols for IP-Networked Storage. Technical Report TR03-39, Department of Compute Science, University of Massachusetts, Amherst, September 2003.

[10] K K. Ramakrishnan and J Emer. Performance Analysis of Mass Storage Service Alternatives for Distributed Systems. *IEEE Trans. on Software Engineering*, 15(2):120–134, February 1989.

[11] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, June 1985.

[12] P Sarkar and K Voruganti. IP Storage: The Challenge Ahead. In *Proceedings of the 19th IEEE Symposium on Mass Storage Systems, College Park, MD*, April 2002.

[13] K. Shirriff and J. Ousterhout. A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System. In *Proceedings of the Winter 1992 USENIX Conference*, pages 315–331, January 1992.

[14] Performance Comparison of iSCSI and NFS IP Storage Protocols. Technical report, TechnoMages, Inc.

A Versatile and User-Oriented Versioning File System

Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok
Stony Brook University

Abstract

File versioning is a useful technique for recording a history of changes. Applications of versioning include backups and disaster recovery, as well as monitoring intruders' activities. Alas, modern systems do not include an automatic and easy-to-use file versioning system. Existing backup solutions are slow and inflexible for users. Even worse, they often lack backups for the most recent day's activities. Online disk snapshotting systems offer more fine-grained versioning, but still do not record the most recent changes to files. Moreover, existing systems also do not give individual users the flexibility to control versioning policies.

We designed a lightweight user-oriented versioning file system called *Versionfs*. Versionfs works with any file system and provides a host of user-configurable policies: versioning by users, groups, processes, or file names and extensions; version retention policies and version storage policies. Versionfs creates file versions automatically, transparently, and in a file-system portable manner—while maintaining Unix semantics. A set of user-level utilities allow administrators to configure and enforce default policies: users can set policies within configured boundaries, as well as view, control, and recover files and their versions. We have implemented the system on Linux. Our performance evaluation demonstrates overheads that are not noticeable by users under normal workloads.

1 Introduction

Versioning is a technique for recording a history of changes to files. This history is useful for restoring previous versions of files, collecting a log of important changes over time, or to trace the file system activities of an intruder. Ever since Unix became popular, users have desired a versatile and simple versioning file system. Simple mistakes such as accidental removal of files (the infamous "`rm *`" problem) could be ameliorated on Unix if users could simply execute a single command to undo such accidental file deletion.

CVS is one of the most popular versioning tools [2]. CVS allows a group of users to record changes to files in a repository, navigate branches, and recover any version officially recorded in a CVS repository. However, CVS does not work transparently with all applications.

Another form of versioning is backup tools such as Legato's Networker [15] or Veritas's Backup Exec [27] and FlashSnap [28]. Modern backup systems include specialized tools for users to browse a history of file changes and to initiate a recovery of file versions. However, backup systems are cumbersome to use, run slowly, and they do not integrate transparently with all user applications. Worse, backup periods are usually set to once a day, so potentially all file changes within the last 24-hour period are not backed up.

Another approach is to integrate versioning into the file system [4, 17, 24, 25]. Developing a native versioning file system from scratch is a daunting task and will only work for one file system. Instead, we developed a stackable file system called *Versionfs*. A stackable file system operates at the highest possible layer inside the OS. Versionfs can easily operate on top of any other file system and transparently add versioning functionality without modifying existing file system implementations or native on-media structures. Versionfs monitors relevant file system operations resulting from user activity, and creates backup files when users modify files. Version files are automatically hidden from users and are handled in a Unix-semantics compliant manner.

To be flexible for users and administrators, Versionfs supports various *retention* and *storage* policies. Retention policies determine how many versions to keep per file. Storage policies determine how versions are stored. We define the term *version set* to mean a given file and all of its versions. A user-level dynamic library wrapper allows users to operate on a file or its version set without modifying existing applications such as `ls`, `rm`, or `mv`. Our library makes version recovery as simple as opening an old version with a text editor. All this functionality removes the need to modify user applications and gives users a lot of flexibility to work with versions.

We developed our system under Linux. Our performance evaluation shows that the overheads are not noticeable by users under normal workloads.

The rest of this paper is organized as follows. Section 2 surveys background work. Section 3 describes the design of our system. We discuss interesting im-

plementation aspects in Section 4. Section 5 evaluates Versionfs's features, performance, and space utilization. We conclude in Section 6 and suggest future directions.

2 Background

Versioning was provided by some early file systems like the Cedar File System [7] and 3DFS [13]. The main disadvantage of these systems was that they were not completely transparent. Users had to create versions manually by using special tools or commands. CVS [2] is a user-land tool that is used for source code management. It is also not transparent as users have to execute commands to create and access versions.

Snapshotting or check-pointing is another approach for versioning, which is common for backup systems. Periodically a whole or incremental image of a file system is made. The snapshots are available on-line and users can access old versions. Such a system has several drawbacks, the largest one being that changes made between snapshots can not be undone. Snapshotting systems treat all files equally. This is a disadvantage because not all files have the same usage pattern or storage requirements. When space must be recovered, whole snapshots must be purged. Often, managing such snapshot systems requires the intervention of the administrator. Snapshotting systems include AFS [12], Plan-9 [20], WAFL [8], Petal [14], Episode [3], Venti [21], Spiralog [10], a newer 3DFS [22] system, File-Motel [9], and Ext3COW [19]. Finally, programs such as `rsync`, `rdiff`, and `diff` are also used to make efficient incremental backups.

Versioning with copy-on-write is another technique that is used by Tops-20 [4], VMS [16], Elephant File System [24], and CVFS [25]. Though Tops-20 and VMS had automatic versioning of files, they did not handle all operations, such as rename, etc.

Elephant is implemented in the FreeBSD 2.2.8 kernel. Elephant transparently creates a new version of a file on the first write to an open file. Elephant also provides users with four retention policies: keep one, keep all, keep safe, and keep landmark. Keep one is no versioning, keep all retains every version of a file, keep safe keeps versions for a specific period of time but does not retain the long term history of the file, and keep *landmark* retains only important versions in a file's history. A user can mark a version as a landmark or the system uses heuristics to mark other versions as landmark versions. Elephant also provides users with the ability to register their own space reclamation policies. However, Elephant has its own low-level FFS-like disk format and cannot be used with other systems. It also lacks the ability to provide an extension list to be included or excluded from versioning. User level applications have to be modified to access old versions of a file.

CVFS was designed with security in mind. Each individual write or small meta-data change (e.g., atime updates) are versioned. Since many versions are created, new data structures were designed so that old versions can be stored and accessed efficiently. As CVFS was designed for security purposes, it does not have facilities for the user to access or customize versioning.

NTFS version 5, released with Windows 2000, provides *reparse points*. Reparse points allow applications to enhance the capabilities of the file system without requiring any changes to the file system itself [23]. Several backup products, such as Storeactive's LiveBackup [26], are built using this feature. These products are specific to Windows, whereas Versionfs uses stackable templates and can be ported to other OSes easily.

Clearly, attempts were made to make versioning a part of the OS. However, modern operating systems still do not include versioning support. We believe that these past attempts were not very successful as they were not convenient or flexible enough for users.

3 Design

We designed Versionfs with the following four goals:

Easy-to-use: We designed our system such that a single user could easily use it as a personal backup system. This meant that we chose to use per-file granularity for versions, because users are less concerned with entire file system versioning or block-level versioning. Another requirement was that the interface would be simple. For common operations, Versionfs should be completely transparent.

Flexibility: While we wanted our system to be easy to use, we also made it flexible by providing the user with options. The user can select minimums and maximums for how many versions to store and additionally how to store the versions. The system administrator can also enforce default, minimum, and maximum policies.

Portability: Versionfs provides portable versioning. The most common operation is to read or write from the *current version.* We implemented Versionfs as a kernel-level file system so that applications need not be modified for accessing the current version. For previous version access, we use library wrappers, so that the majority of applications do not require any changes. Additionally, no operating system changes are required for Versionfs.

Efficiency: There are two ways in which we approach efficiency. First, we need to maximize current version performance. Second, we want to use as little space as possible for versions to allow a deeper history to be kept. These goals are often conflicting, so we provide various storage and retention policies to users and system administrators.

We chose a stackable file system to balance efficiency and portability. Stackable file systems run in kernel-space and perform well [30]. For current version access, this results in a low overhead. Stackable file systems are also portable. System call interfaces remain unchanged so no application modifications are required.

The rest of this section is organized as follows. Section 3.1 describes how versions are created. Section 3.2 describes storage policies. Section 3.3 describes retention policies. Section 3.4 describes how previous versions are accessed and manipulated. Section 3.5 describes our version cleaning daemon. Section 3.6 describes file system crash recovery.

3.1 Version Creation

In Versionfs, the *head*, or current, version is stored as a regular file, so it maintains the access characteristics of the underlying file system. This design avoids a performance penalty for reading the current version. The set of a file and all its versions is called a *version set*. Each version is stored as a separate file. For example, the file `foo`'s nth version is named `foo;`$X n$. X is substituted depending on the storage policy used for the version. X could be: "`f`" indicating a full copy, "`c`" indicating a compressed version, "`s`" indicating a sparse version, and "`d`" indicating a versioned directory. We restrict the user from directly creating or accessing files with names matching the above pattern. Previous versioning systems, like the Cedar File System, have used a similar naming convention.

Along with each version set we store a meta-data file (e.g., `foo;i`) that contains the minimum and maximum version numbers as well as the storage method for each version. The meta-data file acts as a cache of the version set to improve performance. This file allows Versionfs to quickly identify versions and know what name to assign to a new version. On version creation, Versionfs also discards older versions according to the retention policies defined in Section 3.3.

Newly created versions are created using a *copy-on-change* policy. Copy-on-change differs from copy-on-write in that writes that do not modify data will not cause versions to be created. The dirty bit that the OS or hardware provides is not sufficient, because it does not distinguish between data being overwritten with the same content or different one.

There are six types of operations that create a version: writes (either through `write` or `mmap` writes), `unlink`, `rmdir`, `rename`, `truncate`, and ownership or permission modifications.

The `write` operations are intercepted by our stackable file system. Versionfs creates a new version if the existing data and the new data differ. Between each open and close, only one version is created. This heuristic approximates one version per save, which is intuitive for users and is similar to Elephant's behavior [24].

The `unlink` system call also creates a version. For some version storage policies (e.g., compression), `unlink` results in the file's data being copied. If the storage policy permits, then `unlink` is translated into a `rename` operation to improve performance. Translating `unlink` to a `rename` reduces the amount of I/O required for version creation.

The `rmdir` system call is converted into a rename, for example `rmdir foo` renames `foo` to `foo;d1`. We only rename a directory that appears to be empty from the perspective of a user. To do this we execute a `readdir` operation to ensure that all files are either versions or version set meta-data files. Deleted directories cannot be accessed unless a user recovers the directory. Directory recovery can be done using the user-level library that we provide (see Section 3.4).

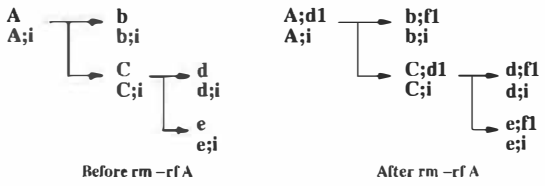


Figure 1: rm -rf on directory A

Figure 1 shows a tree before and after it is removed by `rm -rf`. The `rm` command operates in a depth-first manner. First `rm` descends into `A` and calls `unlink(b)`. To create a version for `b`, Versionfs instead renames `b` to `b;f1`. Next, `rm` descends into `C`, and `d` and `e` are versioned the same way `b` was. Next, `rm` calls `rmdir` on `C`. Versionfs uses `readdir` to check that `C` does not contain any files visible to the user, and then renames it to `C;d1`. Finally, `A` is versioned by renaming it to `A;d1`.

The `rename` system call must create a version of the source file and the destination file. The source file needs a version so that the user can recover it later using the source name. If the destination file exists, then it too must be versioned so its contents are preserved. Whereas we preserve the history of changes to the data in a file, we do not preserve the filename history of a file. This is because we believe that data versioning is more important to users than file-name versioning.

When renaming `foo` to `bar`, if both are regular files, the following three scenarios are possible:

1. **`bar` does not exist:** In this case, we create a version of `foo` before renaming `foo` to `bar`. If both operations succeed, then we create the meta-data file `bar;i`.
2. **`bar` exists:** We first create a version of `bar`. We then create a version of `foo`. Finally, we rename `foo` to `bar`.

3. **`bar` does not exist but `bar;i` exists:** This happens if `bar` has already been deleted and its versions and meta-data files were left behind. In this case, we first create a version for `foo`, then rename `foo` to `bar`. For versioning `bar`, we use the storage policy that was recorded in `bar;i`.

The `rename` system call renames only the head version of a version set. Entire version sets can be renamed using the user-level library we provide (see Section 3.4).

The `truncate` system call must also create a new version. However, when truncating a file `foo` to zero bytes, instead of creating a new version and copying `foo` into the version file, Versionfs renames `foo` to be the version. Versionfs then recreates an empty file `foo`. This saves on I/O that would be required for the copy.

File meta-data is modified when owner or permissions are changed, therefore `chmod` and `chown` also create versions. This is particularly useful for security applications. If the storage policy permits (e.g., sparse mode), then no data is copied.

3.2 Storage Policies

Storage policies define our internal format for versions. The system administrator sets the default policy, which may be overridden by the user. We have developed three storage policies: full, compressed, and sparse mode.

Full Mode Full mode makes an entire copy of the file each time a version is created. As can be seen in Figure 2, each version is stored as a separate file of the form `foo;fN`, where N is the version number. The current, or *head*, version is `foo`. The oldest version in the diagram is `foo;f8`. Before version 8 is created, its contents are located in `foo`. When the page A2 overwrites the page A1, Versionfs copies the entire head version to the version, `foo;f8`. After the version is created, A2 is written to `foo`, then B1, C2, and D2 are written without any further version creation. This demonstrates that in full mode, once the version is created, there is no additional overhead for read or write. The creation of version 9 is similar to the creation of version 8. The first write overwrites the contents of page A2 with the same contents. Versionfs does not create a version as the two pages are the same. When page B2 overwrites page B1, the contents of `foo` are copied to `foo;f9`. Further writes directly modify `foo`. Pages C2, D3, and E1 are directly written to the head version. Version 10 is created in the same way. Writing A2 and B2 do not create a new version. Writing C3 over C2 will create the version `foo;f10` and the head file is copied into `foo;f10`. Finally, the file is truncated. Because a version has already been created in the same session, a new version is not created.

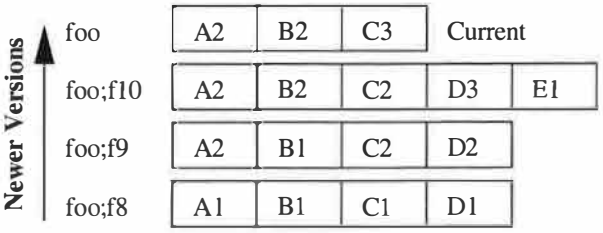


Figure 2: Full versioning. Each version is stored as a complete copy and each rectangle represents one page.

Compress Mode Compress mode is the same as full mode, except that the copies of the file are compressed. If the original file size is less than one block, then Versionfs does not use compression because there is no way to save any space. Compress mode reduces space utilization and I/O wait time, but requires more system time. Versions can also be converted to compress mode offline using our cleaner, described in Section 3.5.

Sparse Mode When holes are created in files (e.g., through `lseek` and `write`), file systems like Ext2, FFS, and UFS do not allocate blocks. Files with holes are called *sparse* files. Sparse mode versioning stores only block deltas between two versions. Only the blocks that change between versions are saved in the version file. It uses sparse files on the underlying file system to save space. Compared to full mode, sparse mode versions reduce the amount of space used by versions and the I/O time. The semantics of sparse files are that when a sparse section is read, a zero-filled page is returned. There is no way to differentiate this type of page with a page that is genuinely filled with zeros. To identify which pages are holes in the sparse version file, Versionfs stores sparse version meta-data information at the end of the version file. The meta-data contains the original size of the file and a bitmap that records which pages are valid in this file. Versionfs does not preallocate intermediate data pages, but does leave logical holes. These holes allow Versionfs to backup changed pages on future writes without costly data-shifting operations [29].

Two important properties of our sparse format are: (1) a normal file can be converted into a sparse version by renaming it and then appending a sparse header, and (2) we can always discard tail versions because reconstruction only uses more recent versions.

To reconstruct version N of a sparse file `foo`, Versionfs first opens `foo;sN`. Versionfs reconstructs the file one page at a time. If a page is missing from `foo;sN`, then we open the next version and attempt to retrieve the page from that version. We repeat this process until the page is found. This procedure always terminates, because the head version is always complete.

Figure 3 shows the contents of `foo` when no versions exist. A meta-data file, `foo;i`, which contains the next

Figure 3: Sparse versioning. Only `foo` exists.

version number, also exists. Figure 4 shows the version set after applying the same sequence of operations as in Figure 2, but in sparse mode.

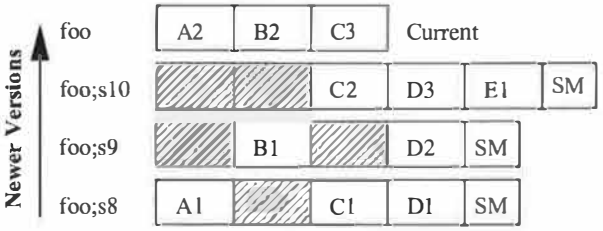


Figure 4: Sparse versioning. Each version stores only block deltas, and each rectangle represents one page. Rectangles with hatch patterns are sparse. Sparse meta-data is represented by the rectangle with "SM" inside.

Versionfs creates `foo;s8` when `write` tries to overwrite page A1 with A2. Versionfs first allocates a new disk block for `foo;s8`, writes A1 to the new block, updates the sparse bitmap and then overwrites A1 with A2 in `foo`. This strategy helps preserve sequential read performance for multi-block files. The other data blocks are not copied to `foo;s8` yet and `foo;s8` remains open. Next, `write` overwrites page B1 with the same data. Versionfs does not write the block to the sparse file because data has not changed. Next, C2 overwrites C1 and Versionfs first writes C1 to the sparse file and then writes C2 to the head version. Versionfs also updates the sparse meta-data bitmap. Page D is written in the same way as page C. The creation of version 9 is similar to version 8. The last version in this sequence is version 10. The pages A2, B2, and C3 are written to the head version. Only C3 differs from the previous contents, so Versionfs writes only C2 to the version file, `foo;s10`. Next, the file is truncated to 12KB, so D3 and E1 are copied into `foo;s10`. The resulting version is shown in Figure 4.

3.3 Retention Policies

We have developed three version retention policies. Our retention policies, as well as Elephant's [24] retention policies, determine how many versions must be retained for a file. However, we provide policies that are different from the ones provided by Elephant. We support the following three retention policies:

Number: The user can set the maximum and minimum number of versions in a version set. This policy is attractive because some history is always kept.

Time: The user can set the maximum and minimum amount of time to retain versions. This allows the user to ensure that a history exists for a certain period of time.

Space: The user can set the maximum and minimum amount of space that a version set can consume. This policy allows a deep history tree for small files, but does not allow one large file to use up too much space.

A version is never discarded if discarding it violates a policy's minimum. The minimum values take precedence over the maximum values. If a version set does not violate any policy's minimum and the version set exceeds any one policy's maximum, then versions are discarded beginning from the tail of the version set.

Providing a minimum and maximum version is useful when a combination of two policies is used. For example, a user can specify that the number of versions to be kept should be 10–100 and 2–5 days of versions should be kept. This policy ensures that both the 10 most recent versions and at least two days of history is kept. Minimum values ensure that versions are not prematurely deleted, and maximums specify when versions should be removed.

Each user and the administrator can set a separate policy for each file size, file name, file extension, process name, and time of day. File size policies are useful because they allow the user to ensure that large files do not use too much disk space. File name policies are a convenient method of explicitly excluding or including particular files from versioning. File extension policies are useful because file names are highly correlated with the actual file type [5]. This type of policy could be used to exclude large multimedia files or regenerable files such as `.o` files. This can also be used to prevent applications from creating excessive versions of unwanted files. For example, excluding ~ from versioning will prevent `emacs` from creating multiple versions of ~ files.

Process name can be used to exclude or include particular programs. A user may want any file created by a text editor to be versioned, but to exclude files generated by their Web browser. Time-of-day policies are useful for administrators because they can be used to keep track of changes that happen outside of business hours or other possibly suspicious times.

For all policies, the system administrator can provide defaults. Users can customize these policies. The administrator can set the minimum and maximum values for each policy. This is useful to ensure that users do not abuse the system. In case of conflicts, administrator-defined values override user-defined values. In case of conflicts between two retention policies specified by a user, the most restrictive policy takes precedence.

3.4 Manipulating Old Versions

By default, users are allowed to read and manipulate their own versions, though the system administrator can turn off read or read-write access to previous versions. Turning off read access is useful because system administrators can have a log of user activity without having the user know what is in the log. Turning off read-write access is useful because users cannot modify old versions either intentionally or accidentally.

Versionfs exposes a set of `ioctl`s to user space programs, and relies on a library that we wrote, *libversionfs* to convert standard system call wrappers into Versionfs `ioctl`s. The libversionfs library can be used as an `LD PRELOAD` library that intercepts each library system call wrapper and directory functions (e.g., `open`, `rename`, or `readdir`). After intercepting the library call, libversionfs determines if the user is accessing an old version or the current version (or a file on a file system other than Versionfs). If a previous version is being accessed, then libversionfs invokes the desired function in terms of Versionfs `ioctl`s; otherwise the standard library wrapper is used. The `LD_PRELOAD` wrapper greatly simplifies the kernel code, as versions are not directly accessible through standard VFS methods.

Versionfs provides the following `ioctl`s: version set stat, recover a version, open a raw version file, and also several manipulation operations (e.g., `rename` and `chown`). Each `ioctl` takes the file descriptor of a parent directory within Versionfs. When a file name is used, it is a relative path starting from that file descriptor.

Version-Set Stat Version-set stat (`vs_stat`) returns the minimum and maximum versions in a version set and the storage policy for each version. This `ioctl` also returns the same information as `stat` for each version.

Recover a Version The version recovery `ioctl` takes a file name F, a version number N, and a destination file descriptor D as arguments. It writes the contents of F's N-th version to the file descriptor D. Providing a file descriptor gives application programmers a great deal of flexibility. Using an appropriate descriptor they can recover a version, append the version to an existing file, or stream the version over the network. A previous version of the file can even be recovered to the head version. In this case, version creation takes place as normal.

This `ioctl` is used by libversionfs to `open` a version file. To preserve the version history integrity, Version files can be opened for reading only. The libversionfs library recovers the version to a temporary file, re-opens the temporary file read-only, unlinks the temporary file, and returns the read-only file descriptor to the caller. After this operation, the caller has a file descriptor that can be used to read the contents of a version.

Open a Raw Version File Opening a raw version returns a file descriptor to an underlying version file. Users are not allowed to modify raw versions. This `ioctl` is used to implement `readdir` and for our version cleaner and converter. The application must first run version-set stat to determine what the version number and storage policy of the file are. Without knowing the corresponding storage policy, the application can not interpret the version file correctly. Through the normal VFS methods, version files are hidden from user space, therefore when an application calls `readdir` it will not see deleted versions. When the application calls `readdir`, libversionfs runs `readdir` on the current version of the raw directory so that deleted versions are returned to user space. The contents of the underlying directory are then interpreted by libversionfs to present a consistent view to user space. Deleted directories cannot be opened through standard VFS calls, therefore we use the raw open `ioctl` to access them as well.

Manipulation Operations We also provide `ioctl`s that `rename`, `unlink`, `rmdir`, `chown`, and `chmod` an entire version set. For example, the version-set chown operation modifies the owner of each version in the version set. To ensure atomicity, Versionfs locks the directory while performing version-set operations. The standard library wrappers simply invoke these manipulation `ioctl`s. The system administrator can disable these `ioctl`s so that previous versions are not modified.

3.4.1 Operational Scenario

All versions of files are exposed by libversionfs. For example, version 8 of `foo` is presented as `foo;8` regardless of the underlying storage policy. Users can read old versions simply by opening them. When a manipulation operation is performed on `foo`, then all files in `foo`'s version set are manipulated.

An example session using libversionfs is as follows. Normally users see only the head version, `foo`.

```
$ echo -n Hello > foo
$ echo -n ", world" >> foo
$ echo '!' >> foo
$ ls
foo
$ cat foo
Hello world!
```

Next, users set an LD PRELOAD to see all versions.

```
$ LD_PRELOAD=libversionfs.so
$ export LD_PRELOAD
```

After using libversionfs as an LD_PRELOAD, the user sees all versions of `foo` in directory listings and can then access them. Regardless of the underlying storage format, libversionfs presents a consistent interface. The second version of `foo` is named `foo;2`. There are no modifications required to standard applications.

```
$ ls
foo  foo;1  foo;2
```

If users want to examine a version, all they need to do is open it. Any dynamically linked program that uses the library wrappers to system calls can be used to view older versions. For example, `diff` can be used to examine the differences between a file and an older version.

```
$ cat 'foo;1'
Hello
$ cat 'foo;2'
Hello, world
$ diff foo 'foo;1'
1c1
< Hello, world!
---
> Hello
```

libversionfs can also be used to modify an entire version set. For example, the standard `mv` command can be used to rename every version in the version set.

```
$ mv foo bar
$ ls
bar  bar;1  bar;2
```

3.5 Version Cleaner and Converter

Using the version-set stat and open raw `ioctl`s we have implemented a version cleaner and converter. As new versions are created, Versionfs prunes versions according to the retention policy as defined in Section 3.3. Versionfs cannot implement time-based policies entirely in the file system. For example, a user may edit a file in bursts. At the time the versions are created, none of them exceed the maximum time limit. However, after some time has elapsed, those versions can be older than the maximum time limit. Versionfs does not evaluate the retention policies until a new version is created. To account for this, the cleaner uses the same retention policies to determine which versions should be pruned. Additionally, the cleaner can convert versions to more compact formats (e.g., compressed versions).

The cleaner is also responsible for pruning directory trees. We do not prune directories in the kernel because recursive operations are too expensive to run in the kernel. Additionally, if directory trees were pruned in the kernel, then users would be surprised when seemingly simple operations take a significantly longer time than expected. This could happen, for example, if a user writes to a file that used to be a directory. If the user's new version needed to discard the entire directory, then the user's simple operation would take an inexplicably long period of time.

3.6 Crash Recovery

In the event of a crash, the meta-data file can be regenerated entirely from the entries provided by `readdir`. The meta-data file can be recovered because we can get the storage method and the version number from the version file names. Versionfs, however, depends on the lower level file system to ensure consistency of files and file names. We provide a high-level file system checker (similar to `fsck`) to reconstruct damaged or corrupt version meta-data files.

4 Implementation

We implemented our system on Linux 2.4.20 starting from a stackable file system template [30]. Versionfs is 12,476 lines of code. Out of this, 2,844 lines were for the various `ioctl`s that we implemented to recover, access, and modify versions. Excluding the code for the `ioctl`s, this is an addition of 74.9% more code to the stackable file-system template that we used. We implemented all the features described in our design, but we do not yet version hard links.

The stackable file system templates we used cache pages both on the upper-level and lower-level file system. We take advantage of the double buffering so that the Versionfs file can be modified by the user (through `write` or `mmap`), but the underlying file is not yet changed. In `commit_write` (used for `write` calls) and `writepage` (used for `mmap`-ed writes), Versionfs compares the contents of the lower-level page to the contents of the upper-level page. If they differ, then the contents of the lower-level page are used to save the version. The memory comparison does not increase system time significantly, but the amount of data versioned, and hence I/O time is reduced significantly.

5 Evaluation

We evaluate our implementation of Versionfs in terms of features as well as performance. In Section 5.1, we compare the features of Versionfs with several other versioning systems. In Section 5.2, we evaluate the performance of our system by executing various benchmarks.

5.1 Feature Comparison

In this section, we describe and compare the features of WAFL [8], Elephant [24], CVFS [25], and Versionfs. We selected these because they version files without any user intervention. We do not include some of the older systems like Tops-20 [4] and VMS [16] as they do not handle operations such as rename, etc. We chose Elephant and CVFS because they create versions when users modify files rather than at predefined intervals. We chose WAFL as it is a recent representative of snapshotting systems. We do not include Venti [21] as it provides a framework that can be used for versioning rather than being a versioning system itself.

	Feature	WAFL	Elephant	CVFS	Versionfs
1	File system implementation method	Disk based	Disk based	Disk based [a]	Stackable
2	Copy-on-Change				✔
3	Comprehensive versioning (data, meta-data, etc.)			✔[b]	
4	Transparent support for compressed versions				✔
5	Landmark retention policy		✔		
6	Number based retention policy		[c]		✔
7	Time based retention policy		✔		✔
8	Space based retention policy				✔
9	Unmodified applications can access previous versions	✔			✔
10	Per-user extension inclusion/exclusion list to version				✔
11	Administrator can override user policies				✔
12	Allows users to register their own reclamation policies		✔		
13	Version tagging		✔		

Table 1: Feature comparison. A check mark indicates that the feature is supported, otherwise it is not.
[a] *Log structured disk-based le system with an NFS interface.*
[b] *Security audit quality versioning.*
[c] *Elephant supports keep all and keep one policies.*

We identified the following thirteen features and have summarized them in Table 1:

1. **File system implementation method:** Versionfs is implemented as a stackable file system, so it can be used with any file system. WAFL, Elephant, and CVFS are disk-based file systems and cannot be used in conjunction with any other file system. CVFS uses log structured disk layout and exposes an NFS interface for accessing the device. The advantage of stacking is that the underlying file system can be chosen based on the users' needs. For example, if the administrator expects a lot of small files to be created in one directory, the user can stack on top of a hash-tree based or tail-merging file system to improve performance and disk usage.
2. **Copy-on-Change:** Versionfs takes advantage of the double-buffering used by stackable templates. On writes, Versionfs compares each byte of the new data with the old data and makes a version only if there are changes. This is advantageous as several applications (e.g., most text editors) rewrite the same data to disk. CVFS and Elephant use copy-on-write for creating new versions, creating versions when there actually is no change.
3. **Comprehensive versioning:** Comprehensive versioning creates a new version on every operation that modifies the file or its attributes. This is particularly useful for security purposes. Only CVFS supports this feature.
4. **Transparent support for compressed versions:** Versionfs supports a policy that allows versions to be stored on disk in a compressed format. In Elephant, a user-land cleaner can compress old files. In Versionfs, users can access the data in compressed files directly by using libversionfs. In Elephant users cannot access the version files directly.
5. **Landmark retention policy:** This policy retains only important or landmark versions in a file's history. Only Elephant supports this policy.
6. **Number based retention policy:** This policy lets the user set upper and lower bounds on the number of versions retained for a file. Versionfs supports this policy. Elephant supports "keep one" policy that retains 0 versions and "keep all" policy that retains infinite versions.
7. **Time based retention policy:** This policy retains versions long enough to recover from errors but versions are reclaimed after a configured period of time. Versionfs supports this policy. Elephant's "keep safe" policy is functionally similar to Versionfs's time based policy.
8. **Space based retention policy:** This policy limits the space used by a file's versions. Only Versionfs supports this policy.
9. **Unmodified application access to previous versions:** Versioning is only truly transparent to the user if previous versions can be accessed without making modifications to user-level applications like `ls`, `cat`, etc. This is possible in Versionfs (through libversionfs) and in WAFL. Elephant and CVFS need modified tools to access previous versions.
10. **Per-user extension inclusion and exclusion lists to version:** It is important that users have the ability to choose the files to be versioned and the policy to be used for versioning, because all files do not need to be versioned equally. Versionfs allows users to specify a list of extensions to be excluded

from versioning. Elephant allows groups of files to have the same retention policies, but does not allow explicit extension lists.

11. **Administrator can override user policies:** Providing flexibility to users means users could misconfigure the policies. It is important that administrators can override or set bounds for the user policies. Versionfs allows administrators to set an upper bound and lower bound on the space, time, and number of versions retained.
12. **Allows users to register their own reclamation policies:** Users might prefer policies other than the system default. Elephant is the only file system that allows users to set up custom reclamation policies.
13. **Version tagging:** Users want to mark the state of a set of files with a common name so that they can revert back to that state in the future. Only Elephant supports tagging.

5.2 Performance Comparison

We ran all the benchmarks on a 1.7GHz Pentium 4 machine with 1GB of RAM. All experiments were located on two 18GB 15,000 RPM Maxtor Atlas hard drives configured as a 32GB software Raid 0 disk. The machine was running Red Hat Linux 9 with a vanilla 2.4.22 kernel. To ensure a cold cache, we unmounted the file systems on which the experiments took place between each run of a test. We ran each test at least 10 times. To reduce I/O effects due to ZCAV we located the tests on a partition toward the outside of the disk that was just large enough for the test data [6]. We recorded elapsed, system, and user times, and the amount of disk space utilized for all tests. We display elapsed times (in seconds) on top of the time bars in all our graphs. We also recorded the wait times for all tests, wait time is mostly I/O time, but other factors like scheduling time can also affect it. It is computed as the difference between the elapsed time and system+user times. We report the wait time where it is relevant or has been affected by the test. We computed 95% confidence intervals for the mean elapsed and system times using the Student-t distribution. In each case, the half-widths of the confidence intervals were less than 5% of the mean. The space used does not change between different runs of the same test. The user time is not affected by Versionfs as it operates only in the kernel. Therefore we do not discuss the user times in any of our results. We also ran the same benchmarks on Ext3 as a baseline.

5.2.1 Configurations

We used the following storage policies for evaluation:

- **FULL:** Versionfs using the full policy.
- **COMPRESS:** Versionfs using the compress policy.
- **SPARSE:** Versionfs using the sparse policy.

We used the following retention policies:

- **NUMBER:** Versionfs using the number policy.
- **SPACE:** Versionfs using the space policy.

For all benchmarks, one storage and one retention configuration were concurrently chosen. We did not benchmark the time retention policy as it is similar in behavior to space retention policy.

5.2.2 Workloads

We ran four different benchmarks on our system: a CPU-intensive benchmark, an I/O intensive benchmark, a benchmark that simulates the activity of a user on a source tree, and a benchmark that measures the time needed to recover files.

The first workload was a build of Am-Utils [18]. We used Am-Utils 6.1b3: it contains over 60,000 lines of C code in 430 files. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 152 new files and 19 new directories. Though the Am-Utils compile is CPU intensive, it contains a fair mix of file system operations, which result in the creation of multiple versions of files and directories. We ran this benchmark with all storage policies and retention policies that we support. However, we report only one set of results as they are nearly identical. This workload demonstrates the performance impact a user sees when using Versionfs under a normal workload. For this benchmark, 25% of the operations are writes, 22% are `lseek` operations, 20.5% are reads, 10% are `open` operations, 10% are `close` operations, and the remaining operations are a mix of `readdir`, `lookup`, etc.

The second workload we chose was Postmark [11]. Postmark simulates the operation of electronic mail servers. It does so by performing a series of file system operations such as appends, file reads, creations, and deletions. This benchmark uses little CPU, but is I/O intensive. We configured Postmark to create 5,000 files, between 512–1,045,068 bytes, and perform 20,000 transactions. For this configuration, 58% of the operations are writes, 37.3% are reads and the remaining are a mix of operations like `open`, `flush`, etc. (We use Tracefs [1] to measure the exact operation mix in the Am-Utils and Postmark benchmarks.) We chose 1,045,068 bytes as the file size as it was the average inbox size on our large campus mail server.

The third benchmark we ran was to copy all the incremental weekly CVS snapshots of the Am-Utils source tree for 10 years onto Versionfs. This simulates the modifications that a user makes to a source tree over a period of time. There were 128 snapshots of Am-Utils, totaling 51,636 files and 609.1MB.

The recover benchmark recovers all the versions of all regular files in the tree created by the copy benchmark. This measures the overhead of accessing a previous version of a file. We end the section with statistics to compare Copy-on-write with Copy-on-change.

5.2.3 Am-Utils Results

Figure 5 and Table 2 show the performance of Versionfs for an Am-Utils compile with the NUMBER retention policy and five versions of each file retained. During the benchmark, all but 15 of the files that are created have less than five versions and the remaining files have between 11 and 1,631 versions created. Choosing five versions as the limit ensures that some of the retention polices are applied. After compilation, the total space occupied by the Am-Utils directory on Ext3 is 33.3MB.

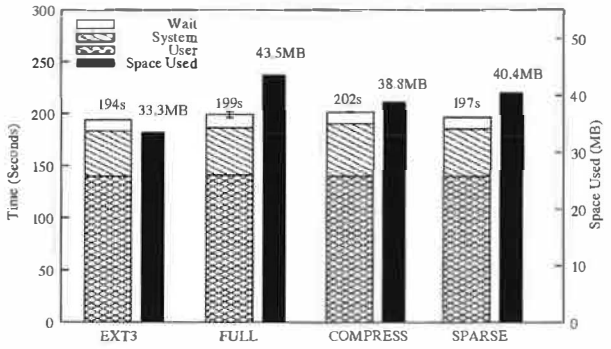


Figure 5: Am-Utils Compilation results. Note: benchmark times use the left scale. Space occupied by each configuration at the end of compilation is represented by the black bars and use the right scale.

	Ext3	Full	Compress	Sparse
Elapsed	193.7s	199.1s	201.6s	196.6s
System	43.4s	45.2s	50.3s	45.4s
Wait	10.8s	12.7s	10.7s	10.8s
Space	33.3MB	43.5MB	38.8MB	40.4MB
Overhead over Ext3				
Elapsed	-	3%	4%	1%
System	-	4%	16%	5%
Wait	-	18%	-1%	0%

Table 2: Am-Utils results.

For FULL, we recorded a 3% increase in elapsed time, a 4% increase in system time, and an 18% increase in wait time over Ext3. The space consumed was 1.31 times that of Ext3. The system time increases as each page is checked for changes before being written to disk. The wait time increases due to extra data that must be written. With COMPRESS, the elapsed time increased by 4%, the system time increased by 16%, and the wait time decreased by 1% over Ext3. The space consumed was 1.17 times that of Ext3. The system time and consequently the elapsed time increase because each version needs to be compressed. COMPRESS has the least wait time, but this gain is offset by the increase in the system time. With SPARSE, the elapsed time increased by 1% and the system time increased by 5% over Ext3. The wait time was the same as Ext3. The space consumed was 1.21 times that of Ext3. SPARSE consumes less disk space than FULL and hence has smaller wait and elapsed time overheads.

5.2.4 Postmark Results

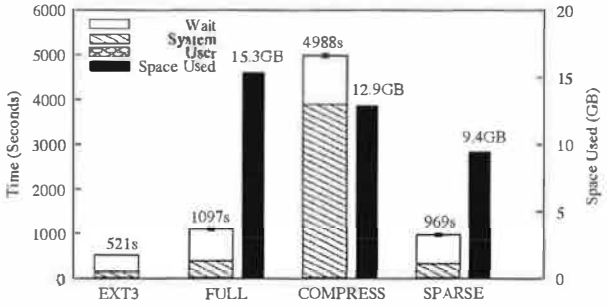


Figure 6: Postmark results. Note: benchmark times use the left scale. Space occupied by each configuration at the end of the test is represented by the black bars and use the right scale.

	Ext3	Full	Compress	Sparse
Elapsed	521s	1097s	4988s	969s
System	128s	368s	3873s	313s
Wait	373s	708s	1093s	634s
Space	0GB	15.34GB	12.85GB	9.43GB
Overhead over Ext3				
Elapsed	-	2.1 ×	9.6×	1.9×
System	-	2.9 ×	30.3×	2.4×
Wait	-	1.9 ×	2.9×	1.7×

Table 3: Postmark results.

Figure 6 and Table 3 show the performance of Versionfs for Postmark with the NUMBER retention policy and nine versions of each file retained. We chose to retain nine versions because eight is the maximum number of versions created for any file and we wanted to retain all the versions of the files. Postmark deletes all the files at the end of the benchmark, so on Ext3 no space is occupied at the end of the test. Versionfs creates versions, so there will be files left at the end of the benchmark.

For FULL, elapsed time was observed to be 2.1 times, system time 2.9 times, and wait time 1.9 times that of Ext3. The increase in the system time is because extra processing has to be done for making versions of files.

The increase in the wait time is because additional I/O must be done in copying large files. The overheads are expected since the version files consumed 15.34GB of space at the end of the test.

For COMPRESS, elapsed time was observed to be 9.6 times, system time 30.3 times and wait time 2.9 times that of Ext3. The increase in the system time is due to the large files being compressed while creating the versions. The wait time increases compared to FULL despite having to write less data. This is because in FULL mode, unlinks are implemented as a rename at the lower level, whereas COMPRESS has to read in the file and compress it. The version files consumed 12.85GB of space at the end of the benchmark.

SPARSE has the best performance both in terms of the space consumed and the elapsed time. This is because all writes in Postmark are appends. In SPARSE, only the page that is changed along with the meta-data is written to disk for versioning, whereas in FULL and COMPRESS, the whole file is written. For SPARSE, elapsed time was 1.9 times, system time 2.4 times, and wait time 1.7 times that of Ext3. The residual version files consumed 9.43GB. The 9.43GB space consumed by SPARSE is the least amount of space consumed for the Postmark benchmark. For a similar Postmark configuration, CVFS, running with a 320MB cache, consumes only 1.3GB, because it uses specialized data structures to store the version meta-data and data. Versionfs cannot optimize the on-disk structures as it is a stackable file system and has no control over the lower-level file system.

9.43GB is more than 9 times the amount of RAM in our system, so these results factor in the effects of double buffering and ensure that our test generated substantial I/O. This benchmark also factors in the cost of `readdir` in large directories that contain versioned files. At the end of this benchmark, 40,053 files are left behind, including versions and meta-data files.

5.2.5 Am-Utils Copy Results

Number Retention Policy Figure 7 and Table 4 show the performance of Versionfs for an Am-Utils copy with the NUMBER retention policy and 64 versions of each file retained. We chose 64 versions as it is median of the number of versions of Am-Utils snapshots we had. Choosing 64 versions also ensures that the effects of retention policies will also be factored into the results. GNU `cp` opens files for copying with the `O_TRUNC` flag turned on. If the file already exists, it gets truncated and causes a version to be created. To avoid this, we used a modified `cp` that does not open files with O_TRUNC flag but instead truncates the file only if necessary at the end of copying. After copying all the versions, the Am-Utils directory on Ext3 consumes 9.9MB.

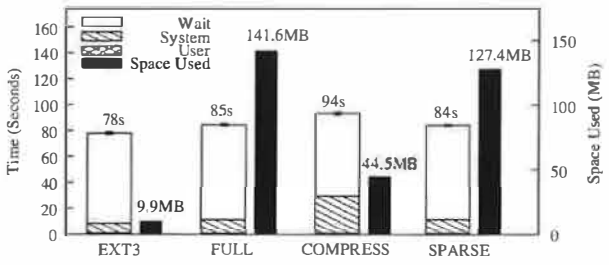


Figure 7: Am-Utils copy results with NUMBER *and 64 versions being retained. Note: benchmark times use the left scale. Space occupied by each configuration at the end of copy is represented by the black bars and use the right scale.*

	Ext3	Full	Compress	Sparse
Elapsed	78.0s	84.6s	93.6s	84.2s
System	7.5s	11.2s	28.9s	11.3s
Wait	69.6s	72.6s	63.8s	72.0s
Space	9.9MB	141.6MB	44.5MB	127.4MB
Overhead over Ext3				
Elapsed	-	8%	20%	8%
System	-	49%	285%	51%
Wait	-	4%	-8%	3%

Table 4: Am-Utils copy results with NUMBER *and 64 versions being retained.*

For FULL, we recorded an 8% increase in elapsed time, a 49% increase in system time, and a 4% increase in wait time over Ext3. FULL consumes 14.35 times the space consumed by Ext3. The system time increases due to two reasons. First, each page is compared and a version is made only if at least one page is different. Second, additional processing must be done in the kernel for making versions.

For COMPRESS, we recorded a 20% increase in elapsed time, a 285% increase in system time, and an 8% decrease in wait time over Ext3. Copy-on-change combined with data compression results in wait time less than even Ext3. This is offset by the increase in the system time due to the compression of version files. As all the versioned files are compressed, the space occupied is the least among all the storage modes and it consumes 4.51 times the space consumed by Ext3.

For SPARSE, we recorded an 8% increase in elapsed time, a 51% increase in system time, and a 3% increase in the wait time over Ext3. Even though SPARSE writes more data than COMPRESS, SPARSE performs better as it does not have to compress the data. The performance of SPARSE is similar to FULL mode since even a small change at the beginning of the file results in the whole file being written. SPARSE consumes 12.91 times the space consumed by Ext3.

Space Retention Policy Figure 8 and Table 5 show the performance of Versionfs for an Am-Utils copy with the SPACE retention policy and each version set having an upper bound of 140KB. We chose 140KB as it is median of the product of the average number of versions per file and the average version file size when all versions are retained. We observed that the number of version files increased for SPARSE and COMPRESS and decreased for FULL. The space occupied by version files decreased. This is because fewer versions of larger files and more versions of smaller files were retained.

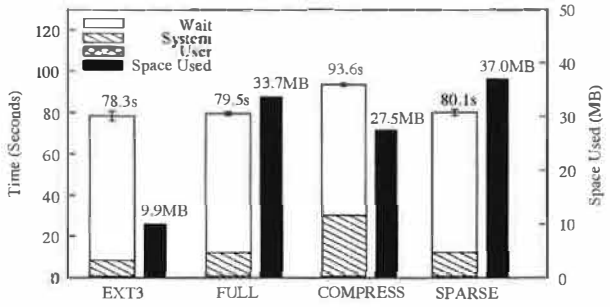


Figure 8: Am-Utils copy results with SPACE *and 140KB being retained per version set. Note: benchmark times use the left scale. Space occupied by each configuration at the end of copy is represented by the black bars and use the right scale.*

	Ext3	Full	Compress	Sparse
Elapsed	78.3s	79.5s	93.6s	80.1s
System	7.7s	11.3s	29.2s	11.6s
Wait	69.9s	67.4s	63.6s	67.6s
Space	9.9MB	33.7MB	27.5MB	37.0MB
Overhead over Ext3				
Elapsed	-	2%	20%	2%
System	-	47%	279%	51%
Wait	-	-4%	-9%	-3%

Table 5: Am-Utils copy results with SPACE *and 140KB being retained per version set.*

For FULL, we recorded a 2% increase in elapsed time, a 47% increase in system time, and an 4% decrease in the wait time compared to Ext3. FULL consumes 3.40 times the space consumed by Ext3.

For COMPRESS, we recorded a 20% increase in elapsed time, a 279% increase in system time, and 9% decrease in the wait time over Ext3. COMPRESS consumes 2.78 times the space consumed by Ext3.

SPARSE has a 2% increase in elapsed time, a 51% increase in system time, and a 3% decrease in wait time over Ext3. SPARSE consumes 3.74 times the space consumed by Ext3. SPARSE takes more space than FULL as it retains more version files. This is because SPARSE stores only the pages that differ between two versions of a file and hence has smaller version files. Consequently, SPARSE packs more files and approaches the 140KB limit per version set more closely.

For all the configurations in this benchmark, we observed that Versionfs had smaller wait times than Ext3. This is because Versionfs can take advantage of Copy-on-change and does not have to write a page if it has not changed. Ext3, however, has to write a page even if it is the same data being overwritten. The system time increases for all configurations as a combined effect of increased `lookup` times and copy-on-change comparisons. System time is the most for COMPRESS as the system has to compress data in addition to the other overheads. Wait time is the least for COMPRESS as the least amount of data is written in this configuration.

5.2.6 Recover Benchmark Results

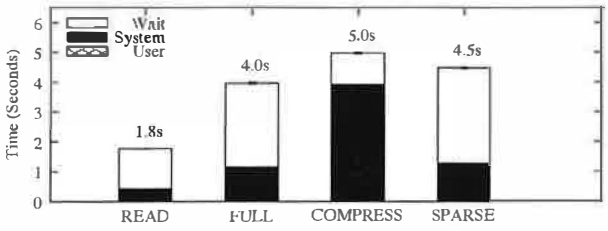


Figure 9: Recover results. READ *is the time to read all files if they were stored as regular files.* FULL, COMPRESS, *and* SPARSE *are the times to recover all versions of all files in the corresponding modes.*

Figure 9 shows the time required to recover all the versions of all files (3,203 versions) that were created by the copy benchmark and the time to read all the versions if they are stored as regular files. The average times to recover a single file in FULL, COMPRESS, SPARSE, and READ were 1.2ms, 1.5ms, 1.4ms, and 0.53ms respectively. The recover times for FULL, COMPRESS, and SPARSE were 2.2, 2.8, and 2.5 times that required to READ, respectively. FULL was the fastest as the recovery time in FULL is the time to copy the required version inside of the kernel. The wait time for FULL increases as the data has to be read from the version file and then written into the recover file. COMPRESS was the slowest as it has to decompress each version. The amount of I/O in COMPRESS is the least as it has to read the least amount of data. However, the time gained in I/O is lost in the system time used for decompressing data. SPARSE mode has the most amount of I/O as it has to reconstruct files from multiple versions as described in Section 3.2.

Benchmark	FULL-COW	FULL-COC	COMPRESS-COW	COMPRESS-COC	SPARSE-COW	SPARSE-COC
Am-Utils copy	370.7MB	141.6MB	168.4MB	44.5MB	370.9MB	127.4MB
Am-Utils compile	43.5MB	43.5MB	38.8MB	38.8MB	41.63MB	40.4MB

Table 6: Copy-on-write (COW) vs. Copy-on-change (COC).

5.2.7 Copy-on-Write vs. Copy-on-Change

Table 6 shows the space consumed by all the storage policies for the Am-Utils copy and Am-Utils compile benchmarks with Copy-on-write and with Copy-on-change. In the Am-Utils copy benchmark, there were considerable savings ranging from 73.6% for COMPRESS to 61.8% for FULL. The savings were good as users generally tend to make minor modifications to the files that copy-on-change can take advantage of.

For the Am-Utils compile benchmark, there is a 2.9% savings in space utilization with copy-on-change in the SPARSE mode and no savings in the FULL and COMPRESS modes. This is because there are more changes to files in Am-Utils compile. Only SPARSE can take advantage of copy-on-change in Am-Utils compile as SPARSE works at a page granularity.

In summary, our performance evaluation demonstrates that Versionfs has an overhead for typical workloads of just 1--4%. With an I/O-intensive workload, Versionfs using SPARSE is 1.9 times slower than Ext3. With all storage policies, recovering a single version takes less than 2ms. Copy-on-change, depending on the load, can reduce disk usage and I/O considerably.

6 Conclusions

The main contribution of this work is that Versionfs allows users to manage their own versions easily and efficiently. Versionfs provides this functionality with no more than 4% overhead for typical user-like workloads. Versionfs allows users to select both what versions are kept and how they are stored through retention policies and storage policies, respectively. Users can select the trade-off between space and performance that best meets their individual needs: full copies, compressed copies, or block deltas. Although users can control their versions, the administrator can enforce minimum and maximum values, and provide users sensible defaults.

Additionally, through the use of libversionfs, unmodified applications can examine, manipulate, and recover versions. Users can simply run familiar tools to access previous file versions, rather than requiring users to learn separate commands, or ask the system administrator to remount a file system. Without libversionfs, previous versions are completely hidden from users.

Finally, Versionfs goes beyond the simple copy-on-write employed by past systems: we implement copy-on-change. Though at first we expected that the comparison between old and new pages would be too expensive, we found that the increase in system time is more than offset by the reduced I/O and CPU time associated with writing unchanged blocks. When more expensive storage policies are used (e.g., compression), copy-on-change is even more useful.

6.1 Future Work

Many applications operate by opening a file using the O_TRUNC option. This behavior wastes resources because data blocks and inode indirect blocks must be freed, only to be immediately reallocated. As the existing data is entirely deleted before the new data is written, versioning systems cannot use copy-on-change. Unfortunately, many applications operate in the same way and changing them would require significant effort. We plan to implement *delayed truncation*, so that instead of immediately discarding truncated pages, they are kept until a write occurs and can be compared with the new data. This can reduce the number of I/O operations that occur.

We will extend the system so that users can register their own retention policies.

Users may want to revert back the state of a set of files to the way it was at a particular time. We plan to enhance libversionfs to support time-travel access.

Rather than remembering what time versions were created, users like to give *tags*, or symbolic names, to sets of files. We plan to store tags and annotations in the version meta-data file.

We will also investigate other storage policies. First, we plan to combine sparse and compressed versions. Preliminary tests indicate that sparse files will compress quite well because there are long runs of zeros. Second, presently sparse files depend on the underlying file system to save space. We plan to create a storage policy that efficiently stores block deltas without any logical holes. Third, we plan to make use of block checksums to store the same block only once, as is done in Venti [21]. Finally, we plan to store only plain-text differences between files. Often when writes occur in the middle of a file, then the data is shifted by several bytes, causing the remaining blocks to be changed. We expect plain-text differences will be space efficient.

7 Acknowledgments

We thank the FAST reviewers for the valuable feedback they provided, as well as our shepherd, Fay Chang. We also thank Greg Ganger and Craig Soules for assisting with extra CVFS benchmarks. This work was partially made possible thanks to an NSF CAREER award EIA-0133589, an NSF award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

References

[1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, San Francisco, CA, March/April 2004.

[2] B. Berliner and J. Polk. Concurrent Versions System (CVS). `www.cvshome.org`, 2001.

[3] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, San Francisco, CA, 1992.

[4] Digital Equipment Corporation. *TOPS-20 User's Guide (Version 4)*, January 1980.

[5] D. Ellard, J. Ledlie, and M. Seltzer. The Utility of File Names. Technical Report TR-05-03, Computer Science Group, Harvard University, March 2003.

[6] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, June 2003.

[7] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, 1988.

[8] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, January 1994.

[9] A.G. Hume. The File Motel – An Incremental Backup System for Unix. In *Summer USENIX Conference Proceedings*, pages 61–72, June 1988.

[10] J. E. Johnson and W. A. Laing. Overview of the Spiralog file system. *Digital Technical Journal*, 8(2):5–14, 1996.

[11] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25, Asilomar Conference Center, Pacific Grove, CA, October 1991. ACM Press.

[13] D. G. Korn and E. Krell. The 3-D File System. In *Proceedings of the USENIX Summer Conference*, pages 147–156, Summer 1989.

[14] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, pages 84–92, Cambridge, MA, 1996.

[15] LEGATO. LEGATO NetWorker. `www.legato.com/products/networker`.

[16] K. McCoy. *VMS File System Internals*. Digital Press, 1990.

[17] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleinman, and S. Owara. SnapMirror: File System Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 117–129, 2002.

[18] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[19] Z. N. J. Peterson and R. C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadat for a Time-Shifting File System. Technical Report HSSL-2003-03, Computer Science Department, The Johns Hopkins University, 2003. `http://hssl.cs.jhu.edu/papers/peterson-ext3cow03.pdf`.

[20] S. Quinlan. A Cached WORM File System. *Software – Practice and Experience*, 21(12):1289–1299, 1991.

[21] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of First USENIX conference on File and Storage Technologies*, pages 89–101, January 2002.

[22] W. D. Roome. 3DFS: A Time-Oriented File Server. In *Proc. of the Winter 1992 USENIX Conference*, pages 405–418, San Francisco, California, 1991.

[23] M. Russinovich. Inside Win2K NTFS, Part 1. `www.winnetmag.com/Articles/ArticleID/15719/pg/2/2.html`, November 2000.

[24] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R.W. Carton, and J. Ofir. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, December 1999.

[25] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata Efficiency in Versioning File Systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, March 2003.

[26] Storactive. Storactive LiveBackup. `www.storactive.com/solutions/liveBackup`.

[27] VERITAS. VERITAS Backup Exec. `www.veritas.com/products/category/ProductDetail.jhtml?productId=bews`.

[28] VERITAS. VERITAS Flashsnap. `www.veritas.com/van/products/flashsnap.html`.

[29] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 289–304, June 2001.

[30] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.

Tracefs: A File System to Trace Them All

Akshat Aranya, Charles P. Wright, and Erez Zadok
Stony Brook University

Abstract

File system traces have been used for years to analyze user behavior and system software behavior, leading to advances in file system and storage technologies. Existing traces, however, are difficult to use because they were captured for a specific use and cannot be changed, they often miss vital information for others to use, they become stale as time goes by, and they cannot be easily distributed due to user privacy concerns. Other forms of traces (block level, NFS level, or system-call level) all contain one or more deficiencies, limiting their usefulness to a wider range of studies.

We developed *Tracefs*, a thin stackable file system for capturing file system traces in a portable manner. Tracefs can capture uniform traces for any file system, without modifying the file systems being traced. Tracefs can capture traces at various degrees of granularity: by users, groups, processes, files and file names, file operations, and more; it can transform trace data into aggregate counters, compressed, checksummed, encrypted, or anonymized streams; and it can buffer and direct the resulting data to various destinations (e.g., sockets, disks, etc.). Our modular and extensible design allows for uses beyond traditional file system traces: Tracefs can wrap around other file systems for debugging as well as for feeding user activity data into an Intrusion Detection System. We have implemented and evaluated a prototype Tracefs on Linux. Our evaluation shows a highly versatile system with small overheads.

1 Introduction

File system traces have been used in the past to analyze the user access patterns and file system software performance. Studies using those traces helped the community devise better software and hardware to accommodate ever-changing computing needs [1, 14, 18]. For example, traces can be used to determine dynamic access patterns of file systems such as the rate of file creation, the distribution of read and write operations, file sizes, file lifetimes, the frequency of each file system operation, etc. The information collected through traces is useful to determine file system bottlenecks. It can help identify typical usage patterns that can be optimized and provide valuable data for improving performance.

Although the primary use of traces had been for file system performance studies [18], two other uses exist: security and debugging. First, file system tracing is useful for security and auditing. Monitoring file system operations can help detect intrusions and assess the damage. Tracing can be conducted file system wide or based on a suspected user, program, or process. Also, file system tracing has the potential for use in computer forensics—to roll back and replay the traced operations, or to revert a file system to a state prior to an attack.

Second, file system tracing is useful for debugging other file systems. A fine-grained tracing facility can allow a file system developer to locate bugs and points of failure. For example, a developer may want to trace just one system call made into the file system, or all calls made by a particular process, or turn tracing on and off dynamically at critical times. A tracing file system that can be easily layered or stacked on top of another file system is particularly suitable for debugging as it requires no modification to the file system or the OS.

Previous tracing systems were customized for a single study [14, 18]. These systems were built either in an ad-hoc manner, or were not well documented in research texts. Their emphasis was on studying the characteristics of file system operations and not on developing a systematic or reusable infrastructure for tracing. After such studies were published, the traces would sometimes be released. Often, the traces excluded useful information for others conducting new studies; information excluded could concern the initial state of the machines or hardware on which the traces were collected, some file system operations and their arguments, pathnames, and more. For example, block-level traces often lack specific information about file system operations and user activity. NFS-level traces lack information about the state of the applications and users running on the clients, or the servers' state. System-call level traces often miss information about how system call activity is translated into multiple actions in the lower layers of the OS. System-call traces also cannot work on NFS servers where no system call activity is seen by the server.

To illustrate typical problems when using past traces, we describe our own experiences. In the past two years, we conducted a study comparing the growth rate of disk

sizes to the growth rate of users' disk space consumption; we required information about the growth rate of various types of files. To determine file types, we used the files' extensions [5]. The traces we were able to access proved unsuitable for our needs. The Sprite [1] and BSD [14] traces were too old to be meaningful for today's fast-changing systems. These two also did not include the full file's pathname or size. The "Labyrinth" and "Lair" passive NFS traces (which were not available at the time we conducted our study) only show patterns as seen over the NFS protocol, lacking client and server information [4]. Eventually, we found two traces which, when combined, could provide us with the data required: the SEER [10] and Roselli [18] traces. Even then, we had to contact the authors of those traces to request additional information about the systems on which those traces were taken. Next, we attempted to draw conclusions from the combination of the two distinct traces, a less-than-ideal situation for precise studies. Finally, to verify our conclusions, we had to capture our own traces and correlate them with past traces. Our experience is not uncommon: much time is wasted because available traces are unsuitable for the tasks at hand.

For traces to be useful for more than one study, they should include all information that could be necessary even years later. To be flexible, the tracing system should allow traces to be captured based on a wide range of fine-grained conditions. To be efficient in time and space, the system should trace only that which is desired, support buffering and compression, and more. To be secure, the trace system should support strong encryption and powerful anonymization. We have designed and implemented such a system, called *Tracefs*.

Tracefs uses a highly flexible and composable set of modules. *Input filters* can efficiently determine what to trace by users, groups, processes, sessions, file system operations, file names and attributes, and more. *Output filters* control trace data manipulations such as encryption, compression, buffering, checksumming—as well as aggregation operators that count frequencies of traced operations. *Output drivers* determine the amount of buffering to use and where the trace data stream should be directed: a raw device, a file, or a local or remote socket. The traces are portable and self-describing to preserve their usefulness in the future. A set of user-level tools can anonymize selective parts of a trace with encryption keys that can unlock desired subsets of anonymized data. Finally, our design decomposes the various components of the system in an extensible manner, to allow others to write additional input or output filters and drivers.

We chose a stackable file system implementation for Tracefs because it requires no changes to the operating system or the file systems being traced. A stackable file system can capture traces with the same ease whether running on individual clients' local file systems (e.g., Ext2 or FFS), on network file system mounts (e.g., NFS or SMBFS), or even directly on NFS file servers.

We developed a prototype of the Tracefs system on Linux. Our evaluation shows negligible time overheads for moderate levels of tracing. Tracefs demonstrates an overhead of less than 2% for normal user operations and 6% for an I/O-intensive workload.

The rest of the paper is organized as follows. Section 2 describes the design of Tracefs. Section 3 discusses interesting implementation aspects. Section 4 presents an evaluation of Tracefs. Section 5 surveys related work. We conclude in Section 6 and discuss future directions.

2 Design

We considered the following six design goals:

Flexibility Flexibility is the most important consideration for a tracing file system like Tracefs. Traditionally, tracing systems have either collected large amounts of traces that are cumbersome to store and parse or were focused at specific areas of study that make them less useful to other studies. We designed Tracefs to support different combinations of traced operations, verbosity of tracing, the trace destination, and security and performance features.

Performance It is essential that tracing does not incur too much performance overhead. Tracing is inherently an expensive operation, since it requires disk or network I/O. When designing tracing mechanisms, tradeoffs have to be made between performance and functionality. In our design, we addressed performance issues through buffering and provisions for limiting the data traced to exactly that which is relevant to each study.

Convenience of Analysis We designed Tracefs to use a simple binary format for generated traces. The traces are self-contained, i.e., they incorporate information about how the trace data is to be parsed and interpreted.

Security One of the uses for a tracing file system is to monitor malicious activity on a system. Hence, it is essential that the generated traces should be protected from attacks or subversion. We incorporated encryption and keyed checksums to provide strong security.

Privacy Public distribution of traces raises concerns about privacy since traces may contain personal information. Such information cannot simply be removed from traces since it is required for correlation. To address privacy concerns, we designed our system to anonymize traces while still retaining information required for correlation. Data fields in traces can be selectively anonymized, providing flexibility in choosing the parts of the traces that need to be anonymized.

Portability We achieved portability through the use of a stackable file system that is available on multiple platforms [21]. A stackable file system allows us to easily trace any underlying file system. Since Tracefs is implemented as a kernel module, no kernel modifications are required to enable tracing.

In Section 2.1 we describe the component architecture of Tracefs, and in Sections 2.2–2.5 we discuss each component in detail. In Section 2.6 we describe the trace file structure. In Section 2.7 we describe how traces are anonymized. In Section 2.8 we discuss usage scenarios.

2.1 Component Architecture

Tracefs is implemented as a stackable file system that can be stacked on top of any underlying file system. Figure 1 shows that Tracefs is a thin layer between the *Virtual File System* (VFS) and any other file system. File-system-related system calls invoke VFS calls which in turn invoke an underlying file system. When Tracefs is stacked on top of another file system, the VFS calls are intercepted by Tracefs before being passed to the underlying file system. Before invoking the underlying file system, Tracefs calls hooks into one or more tracers that trace the operation. Another hook is called at the end of the operation to trace the return value.

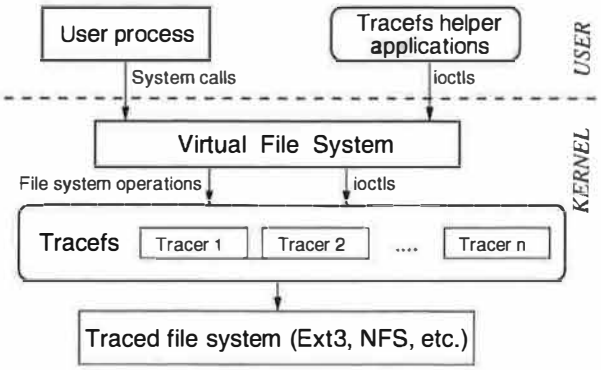


Figure 1: Architecture of Tracefs as a stackable file system. Tracefs intercepts operations and invokes hooks into one or more tracers before passing the operations to the underlying file system.

The use of stacking has several inherent advantages for tracing. Tracefs can be used to trace any file system. Moreover, memory-mapped I/O can only be traced at the file-system level. It is also more natural to study file system characteristics in terms of file system operations instead of system calls. Finally, server-side operations of network file systems are performed directly in the kernel, not through system calls.

Figure 2 depicts the high level architecture of our tracing infrastructure. It consists of four major components:

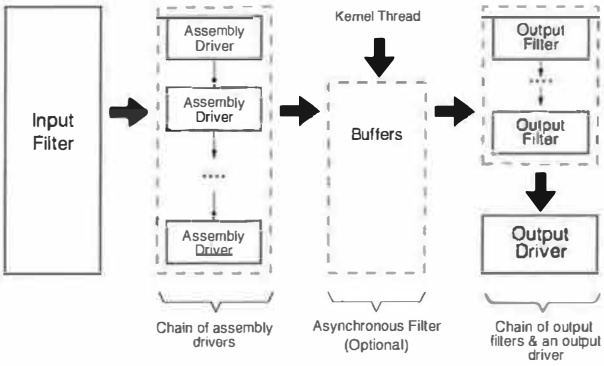


Figure 2: Architecture for a Tracefs tracer. Assembly drivers generate a trace stream that is transformed by output filter s before being written out by an output driver. An optional asynchronous filter buffers the trace stream that is processed by a separate kernel thread.

input filters, assembly drivers, output filters, and output drivers. *Input filters* are invoked using hooks from the file system layer. Input filters determine which operations to trace. *Assembly drivers* convert a traced operation and its parameters into a stream format. *Output filters* perform a series of stream transformations like encryption, compression, etc. *Output drivers* write the trace stream out from the kernel to an external entity, like a file or a socket.

A combination of an input filter, assembly drivers, output filters, and an output driver defines a *tracer*. Tracefs supports multiple tracers which makes it possible to trace the same system simultaneously under different trace configurations.

We have emphasized simplicity and extensibility in designing interfaces for assembly drivers, output filters, and output drivers. Each component has a well-defined API. The APIs can be used to extend the functionality of Tracefs. Writing custom drivers requires little knowledge of kernel programming or file system internals. An output driver or output filter defines five operations: initialize, release, write, flush, and get the preferred block size. An assembly driver requires the implementation of pre-call and post-call stubs for every VFS operation of interest. Including initialization and cleanup, an assembly driver can have up to 74 operations on Linux. Pre-call methods invoke the assembly driver before the actual operation is passed to the lower-level file system; post-call methods invoke the assembly driver after the call to the lower-level file system. For example, an assembly driver that is interested in counting the frequency of file creation and deletion need only implement two methods: CREATE and UNLINK. Custom drivers can be plugged into the existing infrastructure easily.

2.2 Input Filters

We use input filters to specify an expression that determines what operations are traced. For every file system operation, the expression is evaluated to determine if the operation needs to be traced. The user can specify arbitrary boolean expressions built from a set of basic predicates, such as UID, GID, PID, session ID, process name, file name, VFS operation type, system call name, etc. Input filters provide a flexible mechanism for generating specific traces based on the user's requirements.

An input filter is implemented as an in-kernel *directed acyclic graph* (DAG) that represents a boolean expression. We adapted the popular and efficient code for representing expressions from the Berkeley Packet Filter [11] and the PCAP library [8]. We evaluate an input filter against file system objects that were passed as parameters to the call and the current process's task structure. Fields in these structures define the tracing context for evaluating the truth value of the trace expression.

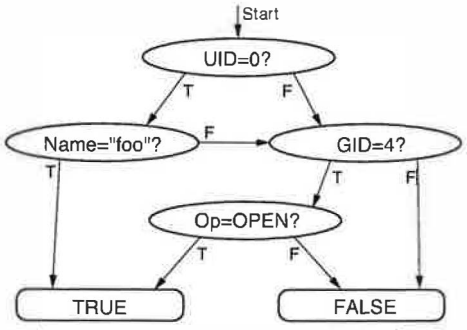


Figure 3: Directed acyclic graph representing the trace condition: $((UID = 0) \wedge (Name = \texttt{foo})) \vee ((GID = 4) \wedge (Op = OPEN))$

Figure 3 shows an example of one such expression and its DAG representation. Each non-terminal vertex in the graph represents a simple predicate, and has two outgoing edges labeled TRUE and FALSE. The graph has two terminal vertices representing the final result of the evaluation. Evaluation starts at the root vertex. At each step, the predicate of the vertex is evaluated, and one of the two outgoing edges is taken based on the result of the evaluation. This procedure is repeated until the graph traversal reaches either of the two terminal vertices.

We chose this approach over a standard tree representation since it is more compact and allows for a simple traversal from the start to a terminal vertex, instead of recursion or complex threaded traversal of trees. It also enables sharing of nodes [11].

To set up tracing, the DAG is constructed in user space and then passed to the kernel using an array. The kernel validates all the data fields, and verifies that the DAG does not contain any cycles. This way the kernel does not need to parse text expressions or construct graphs.

2.3 Assembly Drivers

Once the input filter has determined that a certain operation should be traced, the filter passes the corresponding VFS objects to a series of assembly drivers that determine the content of generated traces. An assembly driver is invoked through a well-defined interface that makes it easy to chain multiple assembly drivers together. We describe two assembly drivers: stream and aggregate.

Stream driver The stream driver is the default assembly driver. It converts the fields of VFS objects into a stream format. The stream driver generates one message for each logged file system operation. We describe the stream format in Section 2.6. The stream driver has various verbosity options that determine how much information is logged. Verbosity options allow the user to choose any combination of the following:

- Fields of VFS objects like `dentry`, `inode`, or `file`, e.g., inode number, link count, and size.
- Return values
- Timestamps with second or microsecond resolution
- Checksum for `read` and `write` data
- Data in `read` and `write` operations
- Process ID, session ID, process group ID
- User ID, group ID
- Process name
- System call number
- File name, file extension

Aggregate driver One of the popular applications of tracing is to collect statistical distributions of operations, instead of actually logging each operation [1, 18]. This has been traditionally performed by post-processing vast amounts of trace data and tallying the number of times each operation was invoked. It is wasteful to log each operation and its parameters since most of the information is discarded in post-processing.

For such applications, we have developed an aggregate driver that keeps track of the number of calls made during a tracing session and records the values at the end of the session. The aggregate driver can be used in conjunction with input filters to determine specific statistical properties of file system operations; for example, to determine the access patterns of individual users.

2.4 Output Filters

Assembly drivers generate a stream of output and feed it to the first of a series of output filters. Output filters perform operations on a stream of bytes and have no knowledge about VFS objects or the trace file format. Each filter transforms the input stream and feeds it to the next filter in the chain. Output filters provide added functionality such as encryption, compression, and checksum calculation. During trace setup, the user can specify the

output filters and their order. The filters can be inserted in any order as they are stream based. The last output filter in the chain feeds the trace stream to an output driver that writes the trace stream to the destination.

Each output filter maintains a default block size of 4KB. We chose a 4KB default, since it is the page size on most systems and it is a reasonably small unit of operation. However, the output filters are designed to operate on streams, therefore, the block sizes can be different for each output filter. The block size can be configured during trace setup.

We now describe four output filters: checksum, compression, encryption, and the asynchronous filter.

Checksum filter A checksum filter is used to verify the integrity of traces. It calculates a block-by-block HMAC-MD5 [17] digest and writes it at the end of each block. It uses a default block size of 4KB that can be overridden. The block size determines how frequently checksums are generated, and therefore, how much overhead checksumming has for the size of the trace file. Each block is also numbered with a sequence number that is included in the digest calculation, so that modification of traces by removal or reordering of blocks can be detected. Each trace file also uses a randomly-generated serial number that is included in each block so that a block in one trace file cannot be replaced with a block with the same sequence number from another file.

A checksum filter ensures that trace files are protected against malicious modifications. Also, since each block has its own digest, we can verify the integrity of each block separately. Even if a few blocks are modified, the unmodified blocks can still be trusted.

Compression filter Traces often contain repeated information. For example, the logged PID, UID, and GID are repeated for each operation performed by a process. Also, the meta-data of the traces, like message identifiers and argument identifiers, is often repeated. As a result, traces lend themselves well to compression.

The compression filter compresses trace data on-the-fly. Compression introduces additional overheads in terms of CPU usage, but it provides considerable savings in terms of I/O and storage space. The compression filter can be used when the size of traces needs to be kept small, or when I/O overheads are large, for example, when traces are recorded over a network.

The compression filter is more efficient when large blocks of data are compressed at once instead of compressing individual messages. We use an input buffer to collect a block of data before compressing it. However, if the compression filter is not the first filter in the chain, its input data is received in blocks and additional input buffering is unnecessary. The compression filter can determine if there is another input filter before it and decide intelligently whether or not to use input buffering. Compression is performed in streaming mode: the compression stream is not flushed until tracing is finished.

Our compression filter uses the zlib library for compression [3]. Zlib is popular, efficient, and provides a tradeoff between speed and compression ratio through multiple compression levels.

Encryption filter The encryption filter secures the contents of generated traces. This ensures that cleartext traces are not written to the disk, which is preferable to encrypting traces offline. We use the Linux CryptoAPI [16]. This allows the use of various encryption algorithms and key sizes.

Asynchronous Filter The asynchronous filter buffers raw trace data from the assembly driver chain and returns immediately. This filter is placed before any other output filter. A separate kernel thread pushes the buffered trace data to the chain of output filters: encryption, compression, etc.—including the final output driver. The asynchronous filter defers CPU-intensive transformations and disk or network I/O. This eliminates expensive operations from the critical path of application execution, which can improve overall performance.

2.5 Output Drivers

Output drivers are similar to output filters in that they operate on a stream of bytes. An output driver writes out the trace stream after it has gone through a series of transformations using output filters. Like output filters, output drivers also employ buffering for efficiency. We now describe two output drivers: file driver and netlink socket driver.

File driver The file device driver writes the output to a regular file, a raw device, or a socket. Since writing to a disk is a slow I/O operation, the file driver uses internal buffers to collect trace data before writing it to the disk. The driver writes the buffer to the disk when the buffer is full. Any data remaining in the buffer is flushed when tracing is completed.

When used in the socket mode, the file driver connects to a TCP socket at a remote location and sends the traces over the network. This mode is useful when local storage is limited and a server with large disk storage is available elsewhere. It is also useful for high-security applications as trace data is never written to the local disk. Additionally, encryption and compression filters can improve security and reduce network traffic.

If Tracefs is used for file system debugging, then the trace file should be kept as current as possible. During code development, it is important that the state of the system is known for the last few events leading up to an error. In such cases, using buffering may not be appropriate. Non-buffered I/O is also applicable to

high-security applications. In hostile environments, the amount of information in memory should be kept to a minimum. Therefore, the file driver also provides a non-buffered mode that writes data immediately. This can be used, for example, to write trace logs to non-erasable tapes. Overall, non-buffered I/O is a tradeoff between latency and performance.

Netlink socket driver Netlink sockets are a feature of the Linux kernel that allows the kernel to set up a special communication channel with a user-level process. Tracefs's netlink socket driver connects to a user level process through a netlink socket and writes the trace stream to the socket. The process can parse the trace stream in real time. For example, the trace stream can be used for dynamic monitoring of file system operations instead of storing for offline post-processing. The trace data can also be used by an intrusion detection system (IDS) to detect unusual file system activity.

2.6 Trace Structure

Traces are generated in a binary format to save space and facilitate parsing. The trace file is composed of two basic building blocks: an argument and a message.

An *argument* represents a field of data in the trace, for example, a PID, UID, timestamp, etc. Each argument is an ⟨ *arg id*, *value* ⟩ or an ⟨ *arg id*, *length*, *value* ⟩ tuple. The *arg id* parameter specifies a unique identifier for the argument. The *length* parameter is only necessary for variable-length fields like file names and process names. The length of constant-length fields can be omitted, thus saving space in the trace. The highest bit of *arg_id* is zero for constant-length fields to indicate that there is no *length* field. Anonymization toggles the highest bit of *arg_id* for constant-length arguments since the length of arguments changes after encryption, due to padding.

A *message* is the smallest unit of data written to the trace. It represents all the data traced for one file system operation. Each message consists of a message identifier, *msg_id*, a length field, and a variable number of arguments. The length field is the length of the entire message. When parsing the trace file, the parser can quickly skip over messages by just reading the *msg id* and *length* fields without parsing the arguments.

The trace file is self-contained in the sense that the meta-data information is encoded within the trace. A trace parser needs to be aware only of the basic building blocks of the trace. The header encodes the message identifiers and argument identifiers with their respective string values. The length of constant-length arguments is also encoded in the header so that it need not be repeated each time the argument occurs in the trace. The length may vary on different platforms and it can be determined from the header when the trace is parsed. Finally, the header also encodes information about the machine the trace was recorded on, the OS version, hardware characteristics like the disk capacity of the mounted file system and the amount of RAM, the input filter, the assembly drivers, the output filters, the output driver for the trace, and other system state information.

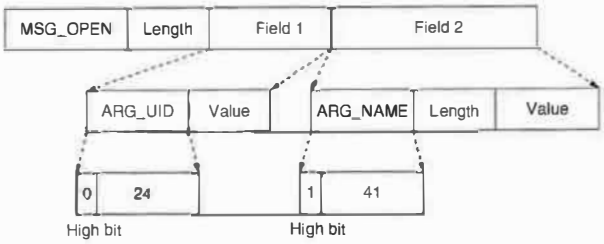


Figure 4: An example of a trace message. Each message contains a message identifier, a length field, and multiple arguments. The highest bit of the argument identifier indicates that the argument has a length field.

Figure 4 shows a partial `open` message. The message has an identifier, `MSG OPEN`, and a length field indicating the entire length of the message. It contains multiple arguments. The figure shows two arguments: `ARG UID` and `ARG NAME`. `ARG UID` is a constant-length argument that does not contain a length field, whereas `ARG NAME` is a variable-length field. Message identifiers are defined for all operations, e.g., `MSG READ` and `MSG WRITE`, and for trace meta-data messages, e.g., `MSG START`. All message identifiers and argument identifiers are encoded in the trace file header; for example, `ARG UID` is encoded as ⟨`24, "ARG_UID"`⟩.

2.7 Anonymization

Distribution of traces raises concerns about security and privacy. Traces cannot be distributed in their entirety as they may reveal too much information about the traced system, especially about user and personal activity [4]. Users are understandably reluctant to reveal information about their files and access patterns. Traces may therefore be anonymized before they are released publicly.

Our anonymization methodology is based on secret-key encryption. Each argument type in the trace is encrypted with a different randomly-generated key. Encryption provides a one-to-one reversible mapping between unanonymized and anonymized fields. Also, different mappings for each field remove the possibility of correlation between related fields, for example UID = 0 and GID = 0 usually occur together in traces, but this cannot be easily inferred from the anonymized traces in which the two fields have been encrypted using different keys. Trace files generated by Tracefs are anonymized offline during post-processing. This allows us to anonymize one source trace file in multiple ways.

Our user-level anonymization tool allows selection of the arguments that should be anonymized. For example,

in one set of traces it may be necessary to anonymize only the file names, whereas in another, UIDs and GIDs may also be anonymized. Anonymized traces can be distributed publicly without encryption keys. Specific encryption keys can be privately provided to someone who needs to extract unanonymized data. Also, the use of encryption makes anonymization more efficient since we do not require lookup tables to map each occurrence of a data field to the same anonymized value. Lookup tables can grow large as the trace grows. Such tables also need to be stored separately if reversible anonymization is required. In contrast, our anonymization approach is *stateless*: we do not have to maintain any additional information other than one encryption key for each type of data anonymized. Finally, we use cipher block chaining (CBC) because cipher feedback (CFB) mode is prone to XOR attacks and electronic code book (ECB) mode has no protection for 8-byte repetitions [19].

2.8 Usage

Tracefs provides user-level tools to setup, start and stop traces. A sample configuration file for trace setup is:

```
{
 { cuid = 0 OR cgid = 1 }
 { stream = { STR_POST_OP | STR_PID |
              STR_UID | STR_TIMESTAMP } }
 { compress;
   filename = "/mnt/trace.log" buf = 262144 }
}
```

The configuration file contains three sections for each tracer: input filter, assembly drivers, and output filters and driver. In this example, the input filter contains two OR-ed predicates. It uses the stream assembly driver, with the parenthesized parameters specifying the verbosity settings. Finally, the output chain consists of the compression filter and the file output driver which specifies the name of the trace file and the buffer size being used. This configuration file is parsed by a tool which calls ioctls to specify the tracer. For the input filter, the tool first constructs a DAG which is then passed to the kernel in a topologically-sorted array. The kernel reconstructs the DAG from this array. If the trace parameters are correct, the kernel returns a unique identifier for the tracer. This identifier can be used later to start and stop tracing using ioctls.

The input filter determines which operations will be traced and under what conditions. The ability to limit traces provides flexibility in applying Tracefs for a large variety of applications. We now discuss three such applications: trace studies, IDSs, and debugging.

Trace Studies When configuring Tracefs for collecting traces for studies, typically all operations will be traced using a simple or null input filter. The stream assembly driver will trace all arguments. The output driver will typically be a file with buffered asynchronous writes for maximum performance.

Intrusion Detection Systems An IDS is configured with two tracers. The first tracer is an aggregate counter that keeps track of how often each operation is executed. This information is periodically updated and a monitoring application can raise an alarm in case of abnormal behavior. The second tracer creates a detailed operation log. In case of an alarm, the IDS can read this log and get detailed information of file system activity. An IDS needs to trace only a few operations. The output filter includes checksumming and encryption for security. The trace output is sent over a socket to a remote destination, or written to a non-erasable tape. Additionally, compression may be used to limit network traffic.

To defeat denial of service attacks, a QoS output filter can be implemented. Such a filter can effectively throttle file system operations, thus limiting resource usage.

Debugging For debugging file systems, Tracefs can be used with a precise input filter, which defines only the operations that are a part of a sequence of operations known to be buggy. Additionally, specific fields of file system objects can be traced, (e.g., the inode number, link count, dentry name, etc.). No output filters need to be used because security and storage space are not the primary concern and the trace file should be easy to parse. The file output driver is used in unbuffered synchronous mode to keep the trace output as up-to-date as possible.

3 Implementation

We developed a prototype of Tracefs as a kernel module for Linux 2.4.20 based on a stackable file system template [21]. Tracefs is 9,272 lines of code. The original stacking template was 3,659 lines of code. The netlink socket output driver is not yet implemented.

We now describe three interesting aspects of our implementation: system-call based filtering, file-name based filtering and asynchronous filter.

System call based filtering To support system call based filtering, we needed to determine which system call invoked the file system operation. System call numbers are lost at the file system level. All other information that we log is available either through function parameters, or globally (e.g., the current running task structure contains the PID, current UID, etc.). System call filtering requires a small patch to the kernel. The patch is required only for this additional functionality. All other features are still available without any modifications to the Linux kernel.

We added an extra field, `sys call`, to `struct task struct`, the structure for tasks. All system calls

are invoked through a common entry point parameterized by the system call number. We added four lines of assembly code to the system call invocation path to set the system call number in the task structure before entry and then reset it on exit. In Tracefs, we can filter based on the system call number by comparing the `sys call` field of the current process's task structure with a bitmap of system calls being traced. We can also record the system call number for each operation so that file system operations can be correlated with system calls.

File name based filtering Implementation of file name based filtering posed a challenge when developing Tracefs. The name of the file is available only in the `dentry` object of a file, not in the `inode`. There are some VFS operations that do not pass the `dentry` object to the file system. Even in cases when the `dentry` is available, comparison of file names might require expensive string matching.

To implement file name and file extension based tracing, we developed a file name cache that stores all inode numbers for inodes that match a specified name or extension. In case of hard links, the inode number is present in every name group that the names of the file satisfy.

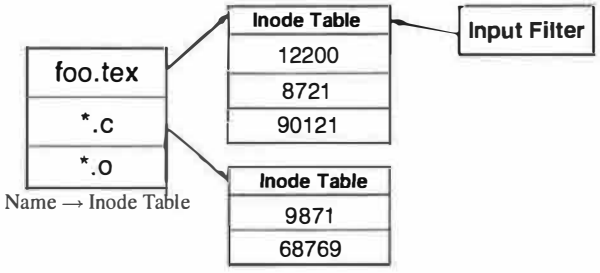


Figure 5: The file name cache for tracing based on file names and extensions. The first level table maps a name to an inode number table that stores inode numbers with that name. An input filter has a direct reference to the inode number table.

Figure 5 shows the structure of the name cache. The figure shows that the name cache is implemented as a two-level hash table. The first level table maps the name or extension to a second level table that stores all the inode numbers that satisfy the rule. The input filter has a direct reference to an inode table. A file name predicate is evaluated by simply looking up the inode number in the inode table. We create a new entry in the table for a newly created inode if its name satisfies any of the rules; the entry is removed when the inode is flushed from memory. Multiple input filters share the same inode table if they are evaluating the same file name predicate. In the figure, `foo.tex` is one of the file names being traced. This will trace operations on all files named `foo.tex`. The entry in the first level table for `foo.tex` points to an inode table that contains three entries, one for each inode that has the name `foo.tex`. The input filter that has a file name predicate for `foo.tex` directly refers to the inode table for predicate evaluation.

Asynchronous filter The asynchronous filter allows Tracefs to move expensive operations out of the critical execution path, as described in Section 2.4. The filter maintains a list of buffers and two queues: the *empty queue* and the *full queue*. The empty queue contains the list of buffers that are currently free and available for use. The full queue contains buffers that are filled and need to be written out. The buffer size and the number of buffers can be configuring during trace setup. The main execution thread picks the first available buffer from the empty queue and makes it the current buffer. The trace stream is written into this current buffer until it is filled up, at which point it is appended to the full queue and the next empty buffer is picked up. A separate kernel thread is spawned at tracing startup; it gets a buffer from the full queue, writes the buffer to the next output filter, and inserts the emptied-out buffer into the empty queue. Each queue has a counting semaphore that indicates the number of buffers available in it. Both threads wait on the respective queue semaphores when they are zero.

4 Evaluation

We evaluated the performance of Tracefs on a 1.7GHz Pentium 4 machine with 1.2GB of RAM. All experiments were conducted on a 30GB 7200 RPM Western Digital Caviar IDE disk formatted with Ext3. To isolate performance characteristics, the traces were written to a separate 10GB 5400 RPM Seagate IDE disk. To reduce ZCAV effects, the tests took place in a separate partition toward the outside of the disk, and the partition size was just large enough to accommodate the test data [7]. The machine ran Red Hat Linux 9 with a vanilla 2.4.20 kernel. Our kernel was modified with a patch for tracing by system call number, so that we could include system call numbers in the trace. However, the results obtained without the optional kernel patch were indistinguishable from those we discuss here. To ensure cold cache, between each test we unmounted the file system on which the experiments took place and to which the traces were written. All other executables and libraries (e.g., compilers) were located on the root file system. We ran all tests at least ten times, and computed 95% confidence intervals for the mean elapsed, system, and user times using the Student-t distribution. In each case, the half-width of the interval was less than 5% of the mean.

4.1 Configurations

Tracefs can be used with multiple different configurations. In our benchmarks, we chose indicative configu-

rations to evaluate performance over the entire spectrum of available features. We conducted the tests at the highest verbosity level of the stream assembly driver so that we could evaluate worst case performance. We selected seven configurations to isolate performance overheads of each output filter and output driver:

EXT3: A vanilla Ext3, which serves as a baseline for performance of other configurations.

FILE: Tracefs configured to generate output directly to a file using a 256KB buffer for traces. We chose a buffer size of 256KB instead of the default of 4KB because our experiments indicated that a 256KB buffer improves the performance on our test setup by 4–5% over 4KB and there are no additional gains in performance with a larger buffer.

UNBUFFERED-FILE: Tracefs configured to generate output to a file without using internal buffering.

CKSUM-FILE: Tracing with an HMAC-MD5 digest of blocks of 4KB, followed by output to a file.

ENCR-FILE: Blowfish cipher in CBC mode with 128-bit keys, followed by output to a file. We used Blowfish because it is efficient, well understood, and was designed for software encryption [19].

COMPR-FILE: Tracing with zlib compression in default compression mode, followed by output to a file.

CKSUM-COMPR-ENCR-FILE: Tracing with checksum calculation followed by compression, then encryption, and finally output to a file. This represents a worst-case configuration.

We also performed experiments for tracing different operations. This demonstrates the performance with respect to the rate of trace generation. We used the aggregate driver to determine the distribution of file system operations and chose a combination of file system operations that produces a trace whose size is a specific fraction of the full-size trace generated when all operations are traced. We used the following three configurations:

FULL: Tracing all file system operations.

MEDIUM: Tracing only the operations that comprise 40--50% of a typical trace. Our tests with the aggregate driver indicated that `open`, `close`, `read`, `write`, `create`, and `unlink` form 40–50% of all trace messages. We chose these operations because they are usually recorded in all studies. Roselli's study also shows that these operations form 49.5% of all operations [18].

LIGHT: Tracing only the operations that comprise approximately 10% of a typical trace. We chose `open`, `close`, `read`, and `write` for this configuration based on the results of our test. This configuration generates traces with an order of magnitude fewer operations than for FULL tracing.

To determine the computational overhead of evaluating input filters, we executed a CPU-intensive benchmark with expressions of various degrees of complexity, containing 1, 10, and 50 predicates. A one-predicate expression is the simplest configuration and demonstrates the minimum overhead of expression evaluation. We believe that practical applications of tracing will typically use expressions of up to ten predicates. We used 50 predicates to demonstrate worst case performance. The expressions were constructed so that the final value of the expression can be determined only after all predicates are evaluated.

4.2 Workloads

We tested our configurations using two workloads: one CPU intensive and the other I/O intensive. We chose one benchmark of each type so that we could evaluate the performance under different system activity levels. Tracing typically results in large I/O activity. At the same time, our output filters perform CPU-intensive computations like encryption and compression. The first workload was a build of Am-Utils [15]. We used Am-Utils 6.1b3, which contains 430 files and over 60,000 lines of C code. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation. The Am-Utils build process is CPU intensive, but it also exercises the file system because it creates a large number of temporary files and object files. We ran this benchmark with all of the configurations mentioned in Section 4.1.

The second workload we chose was Postmark [9]. We configured Postmark to create 20,000 files (between 512 bytes and 10KB) and perform 200,000 transactions in 200 directories. This benchmark uses little CPU, but is I/O intensive. Postmark focuses on stressing the file system by performing a series of file system operations such as directory lookups, creations, and deletions. A large number of small files being randomly modified by multiple users is common in electronic mail and news servers [9].

4.3 Am-Utils Results

Figure 6 shows the results of the Am-Utils build. The figure depicts system, user, and elapsed times for Am-Utils under different configurations of Tracefs. Each bar shows user time stacked over the system time. The height of the bar depicts the total elapsed time for execution. The error bars show the 95% confidence interval for the test. Each group of bars shows execution times for a particular configuration of output filters and output drivers while bars within a group show times for LIGHT, MEDIUM, and FULL tracing. The leftmost bar in each group shows the execution time on Ext3 for reference.

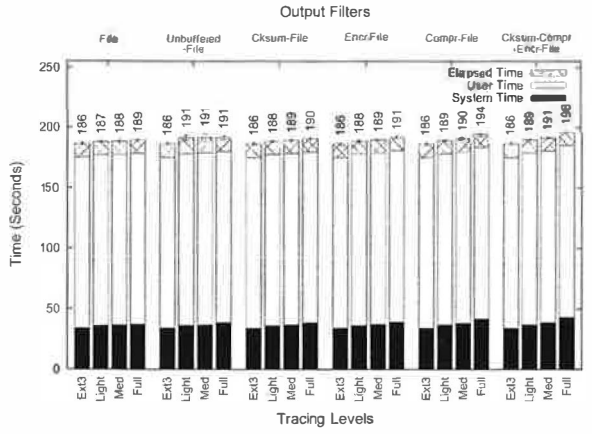


Figure 6: Execution times for an Am-Utils build. Each group of bars represents an output filter configuration under LIGHT, MEDIUM, *and* FULL *tracing. The leftmost bar in each group shows the execution time for Ext3.*

Tracefs incurs a 1.7% elapsed time overhead for FULL tracing when tracing to a file without any output filters. Checksum calculation and encryption introduce additional overheads of 0.7% and 1.3% in elapsed time. Compression results in 2.7% elapsed time overhead. Combining all output filters results in a 5.3% overhead. System time overheads are 9.6–26.8%. The base overhead of 9.6% is due to stacking and handling of the trace stream. CPU intensive stream transformations introduce additional overheads. The low elapsed time overheads indicate that users will not notice a change in performance under normal working conditions.

Under MEDIUM workload, Tracefs incurs a 1.1% overhead in elapsed time when writing directly to a trace file. Checksum calculation and encryption have an additional overhead of less than 1%. Compression has an additional 2.3% overhead. The system time overheads vary from 7.6–13.7%.

Finally, under LIGHT workload, Tracefs incurs less than 1% overhead in elapsed time. The output filters result in an additional overhead of up to 1.1%. System time overheads vary from 6.1–8.6%.

Unbuffered I/O with FULL tracing has an overhead of 2.7%. The system time overhead is 13.7%, an increase of 4.1% over buffered I/O. This shows that buffered I/O provides better performance, as expected.

Overall, these results show that Tracefs has little impact on elapsed time, even with encryption, compression, and checksumming. Among the output filters, compression incurs the highest performance overhead. However, we can see from Figure 8 that the trace file shrinks from 51.1MB to 2.7MB, a compression ratio of 18.8. This indicates that compression is useful for cases where disk space or network bandwidth are limited.

Input Filter Performance We evaluated the performance of input filters on the Am-Utils workload because it is CPU intensive and it gives us an indication of the CPU time spent evaluating expressions under a workload that already has high CPU usage. We tested with input filters containing 1, 10, and 50 predicates. We considered two cases: a FALSE filter that always evaluates to false, and thus never records any trace data; and a TRUE filter that always evaluates to true, and thus constructs the trace data, but writes it to `/dev/null`.

With a FALSE one-predicate input filter, the system time overhead is 4.5%; with a TRUE filter, the overhead is 11.0%. For a ten-predicate input filter, the system time overhead is 8.7% for a false expression and 12.6% for a true expression. Going up to a fifty-predicate filter, the overhead is 16.1% for a FALSE filter and 21.8% for a TRUE filter. In terms of elapsed time, the maximum overhead is 4% with a fifty-predicate filter. Therefore, we can see that Tracefs scales well with complex expressions, and justifies our use of directed acyclic graphs for expression evaluation, as described in Section 2.2.

4.4 Postmark Results

Figure 7 shows the execution times for Postmark. This figure has the same structure as Figure 6 and shows results for the same Tracefs configurations for Postmark.

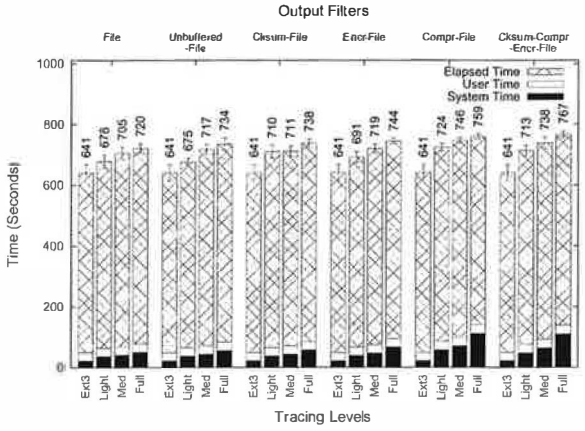


Figure 7: Execution times for Postmark. Each group of bars represents an output filter configuration under LIGHT, MEDIUM, *and* FULL *tracing. The leftmost bar in each group shows execution times for Ext3.*

The figure shows that Tracefs incurs 12.4% overhead in elapsed time for FULL tracing. Encryption introduces another 3.7% overhead, and checksum calculation has an additional overhead of 2.8%. Compression has an overhead of 6.1% in elapsed time. The system time overheads are higher: 132.9% without any output filters, whereas encryption, checksum calculation, and compression have additional overheads of 80.4%, 36.9%, and 291.4%, respectively. However, Figure 7

shows that system time is a small portion of this I/O-intensive benchmark and the performance in terms of elapsed time is reasonable considering the I/O intensive nature of the benchmark.

For MEDIUM tracing, the elapsed time overhead is 10.0%. Encryption, checksum calculation, and compression have additional elapsed time overheads of 2.2%, 0.9%, and 5.2%, respectively. The system time increases by 90.6% without any output filters; encryption, checksum calculation, and compression have additional overheads of 28.7%, 12.8%, and 140.7%, respectively. LIGHT tracing has an overhead ranging from 5.9–11.3% in elapsed time. System time overheads vary from 70.1–122.8%. This shows that selective tracing can effectively limit the computational overheads of transformations. Reducing the trace configuration from FULL to MEDIUM tracing reduces the system time overhead by a factor of 2.1. From FULL to LIGHT tracing, the system time overhead is reduced by a factor of 3.4.

Input Filter Performance We evaluated the performance of input filters for the Postmark workload using a worst-case configuration: a 50-predicate input filter that always evaluates to true, in conjunction with the CKSUM-COMPR-ENCR-FILE configuration of output filters and with FULL tracing. In this configuration, the elapsed time increases from 767 to 780 seconds, an increase of 1.7%, as compared to a one-predicate input filter. This is less than the overhead for Am-Utils because Postmark is I/O intensive.

4.5 Trace File Sizes and Creation Rates

Figure 8 shows the size of trace files (left half) and the file creation rates (right half) for the Am-Utils and Postmark benchmarks. Each bar shows values for FULL, MEDIUM, and LIGHT tracing under a particular configuration. The bar for FILE also shows variation in file size (and rate) for CKSUM-FILE and ENCR-FILE configurations; the values for these two configurations were similar and we excluded them for brevity.

This figure shows that Postmark generates traces at a rate 2.5 times faster than an Am-Utils build. This explains the disparity in overheads between the two benchmarks. Encryption does not increase the file size whereas checksumming increases the file size marginally because checksums are added to each block of data. Trace files achieve a compression ratio in the range of 8–21. The file creation rate decreases as output filters are introduced. However, the rate shows an increase from COMPR-FILE to CKSUM-COMPR-ENCR-FILE since the file size increases because of checksum calculation. The figure also demonstrates how trace files can be effectively reduced in size using input filters.

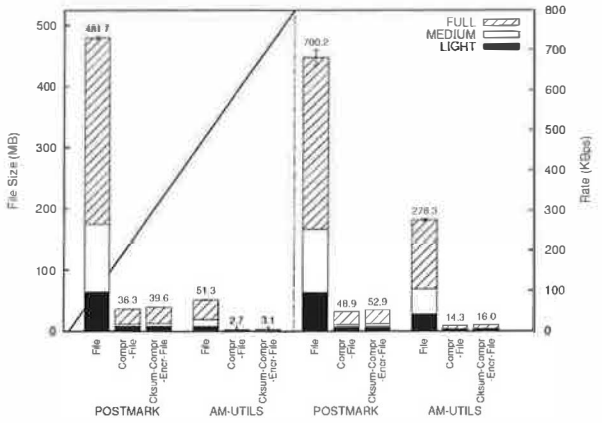


Figure 8: Trace file sizes and creation rates for Postmark and Am-Utils benchmarks. Each bar shows values for FULL, MEDIUM, *and* LIGHT *tracing. The left half depicts file sizes and the right half depicts trace file creation rates*

4.6 Effect of Asynchronous Writes

There are two aspects to Tracefs's performance with respect to asynchronous writes: (1) writing the trace file to disk, and (2) using the asynchronous filter to perform output filter transformations asynchronously. We evaluated Tracefs under the following four configurations:

	Synchronous File System	Asynchronous Filter
WSYNC-TSYNC	yes	no
WSYNC-TASYNC	yes	yes
WASYNC-TSYNC	no	no
WASYNC-TASYNC	no	yes

All benchmarks mentioned previously were performed in WASYNC-TSYNC mode since it is the default configuration for Tracefs and the file system to which the traces were written (Ext2/3). To study the effects of asynchronous writes, we performed two benchmarks: (1) Postmark with a configuration as mentioned in Section 4.2, and (2) OpenClose, a micro-benchmark that performs `open` and `close` on a file in a tight loop using ten threads, each thread performing 300,000 open-close pairs. We chose OpenClose since we determined that general-purpose benchmarks like Postmark perform large I/O on the underlying file system but produce comparatively little I/O for tracing. The OpenClose micro-benchmark is designed to generate large trace data without writing much data to the underlying file system.

Figure 9 shows the system, user, and elapsed times for the two benchmarks under the four configurations. Elapsed time Postmark increases by 3.0% when the traces were written synchronously as compared to asynchronous writes; such an all-synchronous mode is useful for debugging or security applications. For the intensive OpenClose benchmark, synchronous disk writes increase the elapsed time by a factor of 2.3.

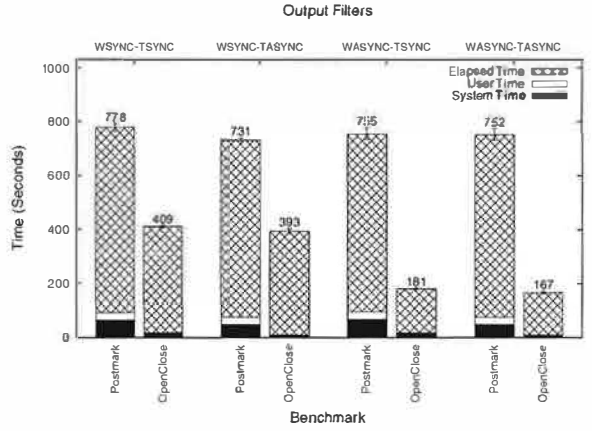


Figure 9: Execution times for Postmark and OpenClose. Bars show system, user, and elapsed times for all combinations of asynchronous filter and trace file writes.

The asynchronous filter lowers the elapsed time for execution of Postmark by 6.0% with synchronous trace file writes. The change with asynchronous disk writes is negligible. For OpenClose, the elapsed time reduces by 3.9% with synchronous disk writes and 7.5% with asynchronous disk writes. The larger improvement is the result of the micro-benchmark stressing the tracing subsystem by generating large traces without actual I/O. The asynchronous filter is useful in such cases.

4.7 Multi-Process Scalability

We evaluated the scalability of Tracefs by executing multiple Postmark processes simultaneously. We configured Postmark to create 10,000 files (between 512 bytes and 10KB) and perform 100,000 transactions in 100 directories for one process, but the workload was evenly divided among the processes by dividing the number of files, transactions, and subdirectories by the number of processes. This ensures that the number of files per directory remains the same. We measured the elapsed time as the maximum of all processes, because this is the amount of time that the work took to complete. We measured the user and system time as the total of the user and system for each process. We executed the benchmark in the FILE configuration with FULL tracing for 1, 2, 4, 8, and 16 processes.

Figure 10 shows the elapsed and system times for Postmark with multiple processes. With a single process, the elapsed time is 383 seconds on Ext3 and 444 seconds with Tracefs. For 16 processes, the elapsed time reduces by a factor of 5.6 on Ext3 and by a factor of 3.3 with Tracefs. This shows that Tracefs scales well with multiple processes, though by a lesser factor, as compared to Ext3. This can be attributed to the serialization of writing traces to disk. The system times remain constant for both Ext3 and Tracefs.

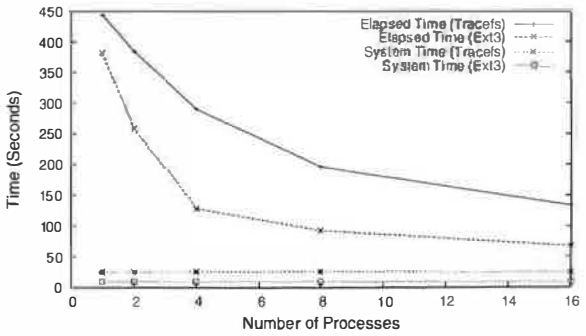


Figure 10: Elapsed and system times for Postmark with Ext3 and Tracefs with multiple processes

4.8 Anonymization Results

Anonymization tests were conducted by anonymizing selected fields of a trace file generated during one run of our Postmark benchmarks under FULL tracing with no output filters. We chose the following five configurations for anonymization:

- Null anonymization. The trace is parsed and rewritten. This serves as the baseline for comparison.
- Process name, file name, and strings anonymized. This accounts for 12.5% of the trace.
- UIDs and GIDs anonymized. This accounts for 17.9% of the trace file.
- Process names, file names, UIDs, GIDs, and strings anonymized. This accounts for 30.4% of the trace.
- Process names, file names, UIDs, GIDs, PIDs and strings anonymized. This accounts for 39.3% of the trace.
- Process names, file names, UIDs, GIDs, PIDs, strings, and timestamps anonymized. This accounts for 48.2% of the trace.

Figure 11 shows the performance of our user-level anonymization tool. The figure shows the size of the anonymized traces in comparison to the original trace file as bars. It also shows the rate of anonymization as a line. The rate of anonymization is the rate at which the unanonymized input trace file can be processed by the anonymization tool.

The leftmost bar in the figure shows the base configuration where no data is anonymized. Each bar shows the size of the anonymized file and the increase in size over the original file. The line shows the rate of anonymization in KBps. The base configuration shows the rate of parsing without any anonymization.

In the figure we see that the rate of anonymization decreases as a larger percentage of data is anonymized, as expected. The rate of anonymization decreases by 30.5% whereas the percentage of anonymized data is increased by 48.2%. The rate of anonymization is limited by the increased I/O required for larger trace files.

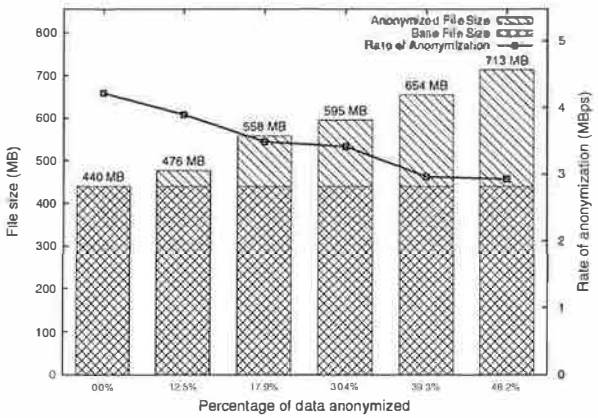


Figure 11: Trace file anonymization rates and increase in file sizes for different portions of traces anonymized. The Y_1 axis shows the scale for file sizes whereas the Y_2 axis shows the scale for the rate of anonymization.

Anonymization increases the trace file size since fields in the trace need to be padded up to the encryption block size. Also, anonymized constant-length fields are converted into variable-length fields since anonymization changes the length of the field. The length of variable-length fields needs to be stored in the trace.

In summary, we evaluated Tracefs's performance using CPU-intensive and I/O-intensive benchmarks under different configurations, and show that Tracefs has acceptable overheads under normal conditions, even with CPU-intensive output filters, or complex input filters.

5 Related Work

In this section we discuss six past trace studies and systems that motivated our design.

File System Tracing Package for Berkeley UNIX In 1984, Zhou et al. implemented a tracing package for the UNIX file system [22]. They instrumented the file operations and process-control-related system calls to log the call and its parameters. Their traces are collected in a binary format and buffered in the kernel before writing to the disk. The package uses a ring of buffers that are written asynchronously using a user-level daemon. The tracing system also switches between trace files so that primary storage can be freed by moving the traces to a tape. The generated binary traces are parsed and correlated into open-close sessions for study. The overhead of tracing is reported up to 10%. The package is comprehensive: it allows tracing of a large number of system calls and logs detailed information about the parameters of each call. However, it provides little flexibility in choosing which calls to trace and the verbosity of the trace. The generated traces require laborious post-processing. Finally, tracing at the system call level makes it impossible to log memory-mapped I/O or to trace network file systems.

BSD Study In 1985, Ousterhout et al. implemented a system for analyzing the UNIX 4.2 BSD file system [14]. In this study, they traced three servers over a period of 2–3 days. This system was implemented by modifying the BSD kernel to trap file-system–related system calls. They chose not to trace reads or writes to avoid generating large traces and consuming too much CPU. Memory-mapped I/O was estimated by logging `execve`. The BSD study was one of the first file system studies and its results influenced the design of future file systems. However, the tracing system used in the study is too specific to be used for other applications. The system traced few operations and other file system activity was inferred, rather than logged. Important file system operations like read, write, lookup, directory reads, and accessing file inodes were not considered.

Sprite Study In 1991, Baker et al. conducted a study on user-level file access patterns on the Sprite operating system [1]. They studied file system activity for the Sprite distributed file system served by four file servers, over four 48-hour periods. In this study, they repeated the analysis of the BSD study. They also analyzed file caching in the Sprite system. They instrumented the Sprite kernel to trace file system calls and periodically feed the data to a user-level logging process. Cache performance was studied by using counters in the kernel that were periodically retrieved and stored by a user-level program. The use of counters provides a lightweight mechanism for statistical evaluation, and we have made similar provisions for aggregate counters in our Tracefs design. However, the Sprite traces are limited to a few file system operations. Like the BSD study, the Sprite study did not record read and write operations to limit CPU and storage overheads. In comparison, Tracefs provides a flexible mechanism to selectively trace any set of file system operations.

Windows NT 4.0 Study In 1998, Vogels conducted a usage study on the Windows NT file system [20]. The purpose of this study was to conduct BSD and Sprite like studies, in the context of changes in computing needs. The usage of components of the Windows NT I/O subsystem was studied. The study was conducted on a set of 45 systems in distinct usage environments. Traces were collected by using a filter driver that intercepts file system requests. The trace driver records 54 I/O request packet (IRP) events on local and remote file system activity covering all major I/O operations. Memory-mapped I/O was also traced. Due to the nature of Windows NT paging, separating actual file system operations from the other VM activity is difficult and must be done during post-processing. This VM activity almost

doubled the size of the traces. Also, daily snapshots of local file systems were taken to record the file system hierarchy. Traces were logged to a remote collection server, and analyzed using data-warehousing techniques.

Roselli Study In 2000, Roselli et al. collected and analyzed file system traces from four separate environments running HP-UX 9.05 and Windows NT 4.0 [18]. HP-UX traces were collected by using the auditing subsystem to record file system events. Additional changes to the kernel were required to trace changes to the current working directory for resolving relative paths in system calls to absolute pathnames. The use of the auditing subsystem provides a mechanism for selectively tracing file system events with minimal changes to kernel code. The system demonstrates that processes frequently use memory mapped I/O; however, tracing of system calls on Unix makes it impossible to determine paging activity that results either from explicit memory-mapped I/O or from loading of executables. Windows NT traces were collected by interposing file system calls using a file system filter driver. Unfortunately, to collect information on memory-mapped operations they needed to interpose not only file system operations, but also system calls. Roselli's filter driver also suffers from problems related to paging activity that are similar to Vogels's.

Passive Network Monitoring Passive network monitoring has been widely used to trace activity on network file systems. Passive tracing is performed by placing a monitoring system on the network that snoops all NFS traffic. The captured packets are converted into a human-readable format and written to a trace file. Post-processing tools are used to parse the trace file and correlate the RPC messages.

Blaze implemented two tools, `rpcspy` and `nfstrace`, to decode RPC messages and analyze NFS operations by deriving the structure of the file system from NFS commands and responses [2].

Ellard et al. implemented a set of tools for anonymization and analysis of NFS traces. These tools capture NFS packets and dump the output in a convenient human-readable format [4, 6]. The traces are generated in a table format that can be parsed using scripts and analyzed with spreadsheets and database software.

Passive tracing has the advantage of incurring no overhead on the traced system. It does not require any modifications to the kernel and can be applied to trace any system that supports the NFS protocol. It also enables the study of an NFS based system as a whole, which is not possible through system call based kernel instrumentation strategies [12, 13]. However, NFS traces are not fully accurate since network packets can be dropped or missed. Passive tracing also does not provide accurate timing information. The NFSv2 and NFSv3 protocols do not have `open` and `close` commands which make it impossible to determine file access sessions accurately. Memory-mapped I/O cannot be distinguished from normal reads and writes using passive NFS tracing. Passive tracing also does not provide an easy mechanism for capturing specific data; large amounts of traces need to be captured and analyzed during post-processing to extract specific information.

Tracefs can be used for monitoring the file system at the server since the tracing is performed at the file system level instead of system call level. Tracefs provides fine-grained control over the specific data to be traced which makes post-processing easier.

6 Conclusions

Our work has three contributions. First, we have created a low-overhead and flexible tracing file system that intercepts operations at the VFS level. Unlike system call tracing, file system tracing records memory-mapped operations. Unlike NFS tracing, file system tracing receives `open` and `close` events. For normal user operations, even with the most verbose traces, our overhead is less than 2%. Our system has several modular components: assembly drivers provide different trace formats or aggregate statistics; output filters perform transformations on the data (e.g., compression or encryption); and output drivers write the traces to various types of media. Low overhead and flexibility makes Tracefs useful for applications where tracing was previously unused: file system debugging, intrusion detection, and more.

Second, Tracefs supports complex input filters that can reduce the amount of trace data generated. Using an input filter to capture open, close, read, and write events, an I/O-intensive workload has only a 6% overhead. Input filters also increase Tracefs's usefulness as a security tool. Tracefs can intercept only suspicious activity and feed it to an IDS.

Third, our trace format is self-contained. No additional information aside from the syntax is required to parse and analyze it. Our trace contains information about the machine that was being traced (memory, processors, disks, etc.) as well as OS information. Rather than use embedded numbers with special meaning, our traces contain mappings of numbers to plaintext operations to ease further analysis (e.g., all system call numbers are recorded at the beginning of the trace).

6.1 Future Work

Our main research focus in this project now shifts to support replaying traces, including selective trace replaying, and replaying at faster or slower rates than the original trace. Replaying traces is useful for several reasons. First, to be able to repeat a sequence of file system operations under different system conditions (e.g., evaluating

with a new file system). Second, for debugging, it is especially useful for validating the reproducibility of a bug in file system code or for changing timing conditions when tracking down a race-condition. Third, for computer forensics it is useful to be able to go back and forth in a trace of suspicious activity, inspecting actions in detail. To support flexible trace replaying, we are investigating what initial state information needs to be recorded in the traces, and possible trace format enhancements.

We are also exploring a few additional aspects of Tracefs. First, we would like to provide a tool to convert binary traces into XML, which can then be processed easily by XML parsers. Second, we are investigating support for capturing lower level disk block information. This information is useful for file system developers to determine optimal on-disk layout. This information is not easily available in a stackable file system.

7 Acknowledgments

We thank the FAST reviewers for the valuable feedback they provided, as well as our shepherd, Yuanyuan Zhou. This work was partially made possible by an NSF CAREER award EIA-0133589, NSF award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of 13th ACM SOSP*, pages 198–212. ACM SIGOPS, 1991.

[2] M. Blaze. NFS Tracing by Passive Network Monitoring. In *Proceedings of the USENIX Winter Conference*, January 1992.

[3] P. Deutsch and J. L. Gailly. RFC 1050: Zlib 3.3 Specification. Network Working Group, May 1996.

[4] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the Annual USENIX Conference on File and Storage Technologies*, March 2003.

[5] D. Ellard, J. Ledlie, and M. Seltzer. The Utility of File Names. Technical Report TR-05-03, Computer Science Group, Harvard University, March 2003.

[6] D. Ellard and M. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the Annual USENIX Conference on Large Installation Systems Administration*, October 2003.

[7] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, June 2003.

[8] LBNL Network Research Group. The TCPDump/Libpcap site. `www.tcpdump.org`, February 2003.

[9] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[10] G. H. Kuenning. *Seer: Predictive File Hoarding for Disconnected Mobile Operation.* PhD thesis, University of California, Los Angeles, May 1997.

[11] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter USENIX Technical Conference*, pages 259–69, January 1993.

[12] A. W. Moore. Operating system and file system monitoring: A comparison of passive network monitoring with full kernel instrumentation techniques. Master's thesis, Department of Robotics and Digital Technology, Monash University, 1998.

[13] A. W. Moore, A. J. McGregor, and J. W. Breen. A comparison of system monitoring methods, passive network monitoring and kernel instrumentation. *ACM SIGOPS Operating Systems Review*, 30(1):16–38, 1996.

[14] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM SOSP*, pages 15–24, Orcas Island, WA, December 1985. ACM.

[15] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[16] H. V. Riedel. The GNU/Linux CryptoAPI site. `www.kerneli.org`, August 2003.

[17] R. L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. Internet Activities Board, April 1992.

[18] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proc. of the Annual USENIX Technical Conference*, pages 41–54, June 2000.

[19] B. Schneier. *Applied Cryptography.* John Wiley & Sons, second edition, October 1995.

[20] W. Vogels. File System Usage in Windows NT 4.0. In *Proceedings of the 17th ACM SOSP*, pages 93–109, December 1999.

[21] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.

[22] S. Zhou, H. Da Costa, and A. J. Smith. A File System Tracing Package for Berkeley UNIX. In *Proceedings of the USENIX Summer Conference*, pages 407–419, June 1984.

HyLog: A High Performance Approach to Managing Disk Layout

Wenguang Wang Yanping Zhao Rick Bunt
Department of Computer Science
University of Saskatchewan
Saskatoon, SK S7N 5A9, Canada
{wang, zhao, bunt}@cs.usask.ca

Abstract

Our objective is to improve disk I/O performance in multi-disk systems supporting multiple concurrent users, such as file servers, database servers, and email servers. In such systems, many disk reads are absorbed by large in-memory buffers, and so disk writes comprise a large portion of the disk I/O traffic. LFS (Log-structured File System) has the potential to achieve superior write performance by accumulating small writes into large blocks and writing them to new places, rather than overwriting on top of their old copies (called Overwrite). Although it is commonly believed that the high segment cleaning overhead of LFS makes it a poor choice for workloads with random updates, in this paper we find that because of the fast improvement of disk technologies, LFS significantly outperforms Overwrite in a wide range of system configurations and workloads (including the random update workload) under modern and future disks.

LFS performs worse than Overwrite, however, when the disk space utilization is very high due to the high cleaning cost. In this paper, we propose a new approach, the Hybrid Log-structured (HyLog) disk layout, to overcome this problem. HyLog uses a log-structured approach for hot pages to achieve high write performance, and Overwrite for cold pages to reduce the cleaning cost. We compare the performance of HyLog to that of Overwrite, LFS and WOLF (the latest improvement on LFS) under various system configurations and workloads. Our results show that, in most cases, Hylog performs comparably to the best of the other three approaches.

1 Introduction

Disk I/O performance is crucial to the performance of many computer systems. With large in-memory buffers, most disk reads can be resolved in memory [18]. As a result, in write-intensive systems, such as a DBMS running OLTP (On-Line Transaction Processing) applications, email servers, and file servers, write requests make up a large portion of the total disk traffic [3, 22]. Since these writes are usually small, most of the disk I/O time is seek time and rotational latency, resulting in less than 10% of the disk maximum bandwidth being utilized [19, 21].

LFS (Log-structured File System) [18, 19] is a disk layout that can achieve superior write performance by writing data to new places in large chunks rather than *overwriting* on top of their old copies. But LFS has to perform segment cleaning to reclaim large contiguous free space for further writes. Previous studies found that this cleaning overhead significantly degrades system performance when the disk space utilization is higher than 50% on a 1991 disk under OLTP workloads [21]. Disk technology has improved dramatically since these studies were published. Using 1991's DEC RZ26 and today's Cheetah X15 36LP as examples, the disk positioning time has decreased from 15ms to 5.6ms (2.7x improvement), while the disk bandwidth has increased from 2.3MB/s to 61MB/s (27x improvement). The disk bandwidth improved 10 times more than the positioning time for these two disks, and this trend is likely to continue [6]. Since LFS was designed to utilize the disk bandwidth, this trend favors the performance of LFS. Whether the cleaning cost of LFS is still prohibitively high under modern and future disks is an unaddressed issue.

In this paper, we use a simple cost model to study the performance of LFS and Overwrite (i.e., the traditional approach that new data are overwritten on top of old copies). Our results show that although the cleaning overhead was expensive for old disks and still accounts for a large amount of disk traffic in modern and future disks, the overall performance of LFS is significantly better than that of Overwrite. LFS loses its advantage to Overwrite only under very high disk space utilization. For example, LFS performs better than Overwrite under uniform random update workload (a pathological workload for LFS that causes the most cleaning cost) when the

disk space utilization is lower than 97% on a year 2003 disk because of its cleaning overhead, assuming 8KB page size and 1MB segment size.

Because of the skewness in page access distribution in real systems [8], most writes are to a small portion of data pages (called *hot* pages), while the other pages (called *cold* pages) are updated infrequently. In LFS, hot pages rarely need to be cleaned because their current copies on the disk are often invalidated by further writes to these pages before the space they occupy is reclaimed by the cleaner. Therefore, most of the cleaning cost comes from cold pages, while most of the high write performance comes from accumulating the writes to hot pages.

We propose a new disk layout called *HyLog* (Hybrid Log-structured). The basic idea underlying HyLog was first mentioned in the conclusions of [12]: "it is not impossible to envision an LFS in which some segments are managed using in-place updating". To our knowledge, no further analyses or experiments have been conducted. HyLog uses a log-structured layout for hot pages to achieve high write performance, and overwrite for cold pages to reduce the cleaning cost. We evaluate the performance of HyLog, Overwrite, LFS, and WOLF (the latest LFS variant [26]) under RAID-0 and RAID-5 disk arrays with concurrent users, a wide range of disk space utilization, and a number of benchmarks and real workloads. We also speculate a disk model five years into the future and study the performance of these disk layouts when using this future disk. Our results show that the performance of HyLog is the most robust among the disk layouts we considered. In most cases, HyLog achieves performance comparable to or better than the best of Overwrite, LFS, and WOLF.

2 Related Work

Many approaches have been proposed to improve disk write performance. NVRAM (non-volatile RAM) is used in many storage systems to cache bursts of writes. Since NVRAM is constrained in size due to its high price, Disk Caching Disk [9] employs a log disk to substitute for NVRAM and achieves similar write performance. The problem with these two approaches is that they achieve high write performance only in systems with many idle periods. Virtual Log is an approach to improving small disk write performance [27] even in systems with no idle periods. But it requires detailed knowledge of the disk layout and the location of the disk head at any moment, which might be difficult to obtain from modern disks [10].

LFS was designed to optimize the write performance of file systems [19]. In LFS, the disk is divided into large fixed-size chunks called *segments*. Writes to data pages are accumulated in memory and written out to free segments. At the same time, the old copies of these pages are invalidated, leaving free space in the segments where they reside. From time to time, *segment cleaning* must be conducted to reclaim the free space in these partially filled segments so that free segments are always available for future writes.

A previous study [21] found that LFS has high cleaning overhead under OLTP-like workloads, where small random writes make up a large portion of the disk I/O requests. Many algorithms have been proposed to reduce the cleaning cost of LFS [2, 14, 16, 19]. But the cleaning cost is still high in systems with high disk space utilization and little idle time. *Freeblock scheduling* [13] has the potential to conduct cleaning without affecting foreground response times, even in a never-idle system. This algorithm relies on detailed knowledge of internal disk activities in order to make correct scheduling decisions, which is difficult for modern disks [10]. PROFS [25] attempts to improve the performance of LFS by placing hot data in the faster zones of the disk and cold data in the slower zones during the cleaning, but this approach does not address the high cleaning cost of LFS. Write Anywhere File Layout (WAFL) [7] and Log-Structured Array [15] use LFS and NVRAM to manage disk layouts. WAFL also maintains multiple snapshots of the file system. Although NVRAM eliminates writes for keeping the metadata integrity and improves write performance, the high cleaning cost of LFS is not addressed.

WOLF [26] is the most recently proposed approach to reducing LFS cleaning overhead. WOLF separates hot and cold pages when they are written to the disk. It usually writes two segments of data to the disk at one time. Pages are sorted based on their update frequencies before being inserted into the segment buffers. The rationale is that the segments containing pages with higher update frequencies will soon become low-utilized since the pages in them are likely to be updated again in a short time, thus reducing the cleaning overhead. This approach works well only when about half of the pages are hot and half are cold, so that they can be written into separate segments. In other cases, WOLF might have little advantage over LFS.

3 Disk Layout Write Cost Model

The extensive use of client and server caching on read traffic makes write performance an important factor in many systems [18]. In fact, write traffic was found to exceed read traffic on some recent file sys-

tem [3] and OLTP workloads [22]. For the purposes of modeling, we assume that the read performance is not affected by different disk layouts, and seek to model the write cost of these disk layouts.

3.1 Assumptions and Definitions

We use a simple disk model with seek time, rotational latency, and transfer bandwidth. The positioning time T_{pos} is the sum of the average seek time and the average rotational latency, i.e., the time for the disk to rotate half a rotation. The transfer bandwidth B is the average sustained bandwidth at which the disk can read/write data. We assume the read bandwidth is the same as the write bandwidth.

We assume that data are stored on the disk in fixed-size pages. The size of each page is P bytes. In LFS, the disk is separated into fixed-size segments, each of which has S pages. The time to read or write a page is T_{pg} and the time to read or write a segment is T_{seg}. We have $T_{pg} = T_{pos} + P/B$ and $T_{seg} = T_{pos} + SP/B$.

3.2 Modelling LFS and Overwrite

3.2.1 Segment I/O Efficiency

One design objective of LFS is to achieve better write performance than Overwrite. This is achieved by writing data in units of segments instead of pages. The *segment I/O efficiency* η represents the saving of disk I/O time for writing one segment over writing S pages of the segment individually. η is defined as

$$\eta = \frac{ST_{pg}}{T_{seg}} = \frac{S(P + T_{pos}B)}{SP + T_{pos}B}. \quad (1)$$

The higher the η, the better the performance of LFS, if other factors are not changed. η is a monolithically increasing function of the segment size S. η is also a monolithically increasing function of $T_{pos}B$, named as the *disk performance product*, which represents the amount of data the disk can transfer during the time to position the disk head. We list the parameters of three high end SCSI disks of different years in Table 1 and show their segment I/O efficiency in Figure 1. Modern disks have much larger η than old disks, implying LFS performs much better on modern disks than on old disks.

When a disk has multiple pending requests from several users, a disk scheduling algorithm is often used to reorder the requests so that the average disk positioning time can be reduced. η decreases with an increase in number of users as a result.

3.2.2 Space Utilization

The disk space utilization is an important factor affecting the performance of LFS [19, 21]. The *disk space utilization* u_d represents the proportion of the disk space occupied by user data. The space utilization of the segments that are selected for cleaning is called the *cleaning space utilization* u, which is the space utilization of the segment with the most free space. Therefore, $u \leq u_d$. More specifically, their relation is given in [15] as

$$u_d = (u - 1)/\ln u. \quad (2)$$

Figure 2 shows that the simulation results match this formula well.

Table 1: Disk Parameters

Brand Name	Year	Positioning Time (ms)	Bandwidth (MB/s)
Cheetah X15 36LP	2003	5.6	61.0
Quantum atlas10k	1999	8.6	20.4
DEC RZ26	1991	15.0	2.3

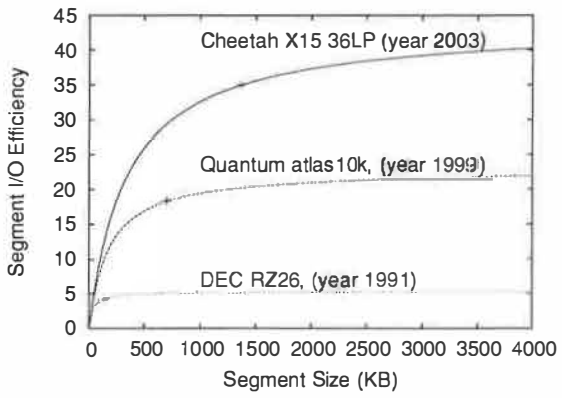


Figure 1: Segment I/O Efficiency of Different Disks. [Page size is 8KB. The small crosses indicate the segment size used for each disk in this paper.]

3.2.3 The Write Cost Model

In Overwrite, each write takes T_{pg} time. Thus the write cost of Overwrite C_{ow} is

$$C_{ow} = T_{pg}$$

To model the write cost of LFS, the segment cleaning overhead must be considered. There are two segment cleaning methods: *cleaning* [19] and *hole-plugging* [14]. We call these variants of LFS *LFS-cleaning* and *LFS-plugging*, respectively.

In LFS-cleaning, some candidate segments for cleaning are first selected, and then these segments

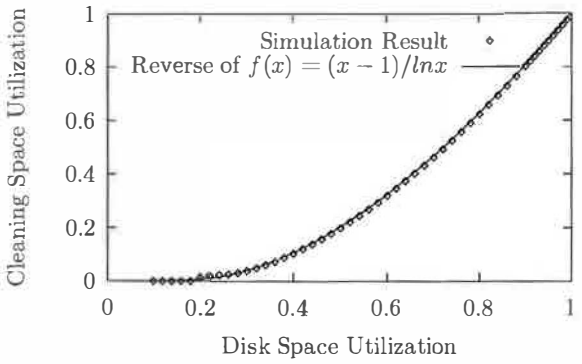


Figure 2: Disk Space and Cleaning Space Utilization.

are read into memory, and their alive pages are written out in segments, after which the free space in these segments is reclaimed. After 1 segment is read, u segment space is written and $1-u$ segment space is freed. Therefore $\frac{1+u}{1-u}$ segment I/O operations are required to free 1 segment space. For the system to be balanced, whenever a segment of user data is written to the disk, a segment of free space is reclaimed by cleaning. Thus LFS requires $1 + \frac{1+u}{1-u} = \frac{2}{1-u}$ segment I/O operations to write one segment of user data. The average time required to write one page in LFS is defined as the write cost $C_{lfscleaning}$.

$$C_{lfscleaning} = \frac{T_{seg}}{S} \cdot \frac{2}{1-u}.$$

In LFS-plugging, some candidate segments are read into memory, and the alive pages of these candidate segments are written out to holes found in other segments so that the space occupied by these candidate segments becomes free. To reclaim one segment of free space, 1 segment read and uS page writes are needed. Therefore, the write cost of LFS-plugging $C_{lfsplugging}$ is defined as the average time required to write one page.

$$C_{lfsplugging} = \frac{1}{S} \cdot (2T_{seg} + uST_{pg}).$$

3.2.4 Performance Comparisons

The performance of these disk layouts can be compared by comparing their write costs. To simplify the write costs, we define the *scaled write cost* by scaling all write costs by $\frac{S}{T_{seg}}$.

$$C'_{ow} = \frac{S}{T_{seg}} C_{ow} = \eta \quad (3)$$

$$C'_{lfscleaning} = \frac{S}{T_{seg}} C_{lfscleaning} = \frac{2}{1-u} \quad (4)$$

$$C'_{lfsplugging} = \frac{S}{T_{seg}} C_{lfsplugging} = 2 + u\eta \quad (5)$$

Note that $C'_{lfscleaning}$ is the same as the traditional write cost of LFS [19]. In [19], the write cost of Overwrite was defined as the reciprocal of the utilized disk bandwidth (i.e., $\frac{T_{pos}B+P}{P}$), which ignores the effect of segment size. Segment size is important to the performance of LFS [14] and is taken into account by C'_{ow}. Figure 3 shows the scaled write cost of the disks listed in Table 1. The relationship between LFS-cleaning and LFS-plugging is consistent with previous studies [14]. Overwrite, LFS-cleaning and LFS-plugging always cross at the same point when $u = 1 - 2/\eta$. Since faster disks have larger η, this cross point happens at higher disk space utilization (e.g., $u = 94\%$ or $u_d = 97\%$ for a year 2003 disk), which means the performance advantage of LFS over Overwrite increases as disk technologies improve. Figure 3 indicates that LFS outperforms Overwrite under such workloads when the cleaning space utilization is below 94% under modern disks. LFS should perform better than what is shown in this figure under more realistic workloads since the cleaning space utilization is lower than that of a uniform random update workload [14]. Therefore, under modern and future disk technologies, the importance of cleaning cost of LFS is much less important than the common belief derived from studies with 10-year-old disks [21].

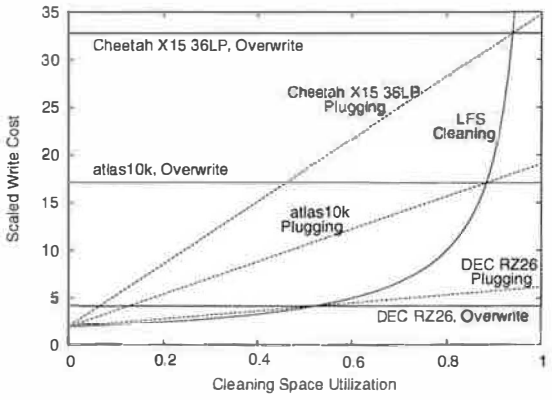


Figure 3: Write Costs of Different Layouts. [Smaller values indicate better performance. The segment size for Cheetah X15 36LP is 1MB, for atlas10k is 512KB, and for DEC RZ26 is 128KB. The selection of segment sizes are discussed in Section 3.3. The write cost of LFS-cleaning for all disks overlaps.]

3.3 Segment Size of LFS

Simulation studies in [14] showed that the segment size is important to the performance of LFS. By experimenting on different disks, the following rules-of-thumb were found [14]:

1. The optimal segment sizes are different for different disks. Only the disk performance product (product of positioning time and transfer bandwidth) matters.
2. Larger segments are required for faster disks. The optimal segment size can be approximated by 4 times the disk performance product.

Equation (1) shows that $T_{pos}B$ is the only disk characteristic that affects η, which is consistent with the first rule-of-thumb. The scaled write costs (Equations (3), (4) and (5)) indicate that the higher the η and the lower the u, and the more advantage LFS can achieve over Overwrite. Figure 1 shows that the larger the segment, the higher the η. However, on one hand, the increase of η is slower with larger segment sizes, while on the other hand, the cleaning space utilization becomes higher with larger segments [14]. Therefore, there is an optimal segment size to achieve the best performance. From Equation (1), we have

$$\lim_{S\to\infty} \eta = \frac{T_{pos}B + P}{P}.$$

Assume that we want to select a segment size so that α proportion of this limit is achieved ($0 < \alpha < 1$). Then

$$\frac{S(T_{pos}B + P)}{T_{pos}B + SP} = \alpha \frac{T_{pos}B + P}{P}.$$

Thus we have

$$S \cdot P = \frac{\alpha}{1-\alpha} T_{pos}B,$$

where $S \cdot P$ is equal to the segment size. If $\alpha = 80\%$, we have

$$S \cdot P = 4T_{pos}B, \tag{6}$$

which is consistent with the second rule-of-thumb. These preferred segment sizes are marked by small crosses in Figure 1. The crosses are close to the "knee" of the curve. In this paper, we use this formula to calculate the segment size and then round it to the closest size in powers of two.

3.4 Multiple Users and RAID

Many systems use disk arrays and have multiple concurrent users. We use N_d to represent the number of disks and N_u to represent the number of users. We assume that users send out requests without think time. When RAID is used, all disks are viewed as one large logical disk. We assume the stripe size is S pages. In RAID-0, the segment size of the logical disk is N_dS; in RAID-5, the segment size of the logical disk is $(N_d-1)S$, because one disk worth of space is used to store parity data. This organization allows segment I/O to utilize all available disk bandwidth and eliminates the write penalty in RAID-5.

3.5 The HyLog Model and Performance Potential

Figure 3 indicates that a small reduction in disk space utilization can significantly reduce segment cleaning cost and improve the performance of LFS. Because of the skewness in page access distribution [8], most writes are to a small portion of hot pages. If only these hot pages are managed by LFS while cold pages are managed by Overwrite, we can dedicate all free space to the hot pages, since Overwrite does not need extra free space. The resulting space utilization for the hot pages would be lower, which implies higher performance for the hot pages. Therefore, the overall performance could exceed both LFS and Overwrite. We call this approach the HyLog layout.

We first give the write cost model of HyLog and then analyze its performance potential.

In HyLog, the disk is divided into fixed-size segments, similar to LFS. A segment is a hot segment (containing hot pages and free pages), a cold segment (containing cold pages and free pages), or a free segment (containing only free pages). The hot segments and free segments form the *hot partition*, and the cold segments form the *cold partition*.

Since LFS-plugging performs worse than LFS-cleaning under low space utilization and worse than Overwrite under high space utilization, including LFS-plugging in HyLog does not bring performance benefit. Therefore, we do not consider LFS-plugging when modeling HyLog. Assume the proportion of hot pages is h ($0 < h < 1$) and the proportion of writes to the hot pages (called *hot writes*) is w ($0 < w < 1$). We use u'_d and u' to represent the disk space utilization and cleaning space utilization of the hot partition, respectively. If all free space is in the hot partition, we have

$$u'_d = \frac{uh}{1-u+uh}. \tag{7}$$

u' can be calculated from u'_d based on Equation (2). The scaled write cost of HyLog C'_{hylog} is

$$\begin{aligned} C'_{hylog} &= (1-w)C'_{ow} + wC'_{lfs} \\ &= (1-w)\eta + \frac{2w}{1-u'}. \end{aligned} \tag{8}$$

When h is 0 and 1, the cost of HyLog degrades to Overwrite and LFS, respectively. The proportion of hot writes w is a function of h, which is the CDF of the write frequencies.

For uniformly distributed random access, $w = h$. It was found the CDF of page update frequency in production database workloads follows the $Hill(f_{max}, k, n)$ distribution $Hill(105, 0.528, 0.546)$ [8], which is defined by $f(x) = f_{max} \cdot x^n/(k + x^n)$. Note that these distributions are for page updates before being filtered by the buffer cache. When write through is used (such as in an NFS server), these distributions can also describe the page writes to disks. When write back is used (such as in a database server), the page writes to disks are less skewed (closer to the uniform distribution).

Another distribution commonly used to represent the skewness of data accesses is defined by Knuth [11, p. 400]: $p_i = \frac{1}{N^\theta(n^\theta-(n-1)^\theta)}$, where $i = 1 \ldots N$ and $0 \leq \theta \leq 1$. When $\theta = 1$, this is the uniform distribution. When $\theta = \frac{\log 0.80}{\log 0.20} = 0.1386$, this is the "80-20" rule where 80% references go to 20% pages. We call this distribution $Knuth(a, b)$, where $\theta = \frac{\log 0.01a}{\log 0.01b}$. Figure 4(a) shows the CDF of the above distributions with different parameters.

Figures 4(b) and 4(c) show the scaled write cost of HyLog under these distributions. Equation (2) is used to convert between the disk space utilization and cleaning space utilization. Since this equation works only for uniform random workloads, the results shown in Figure 4(b) and 4(c) are conservative for skewed distributions. With the right number of hot pages, HyLog outperforms both Overwrite and LFS. The higher the skewness of the distribution, the fewer hot pages are required and the more benefit can be achieved. In other words, HyLog has greater performance potential than LFS and Overwrite under high disk space utilization. When the disk space utilization is low, HyLog has limited benefit over LFS.

4 The Design of HyLog

4.1 Design Assumptions

We assume the disk layouts under study (Overwrite, LFS, WOLF, and HyLog) are at the storage level rather than the file system level. Therefore, the allocation and deallocation of data are not known. We assume NVRAM is used by these disk layouts so that small synchronous writes caused by metadata operations are not necessary. Therefore, we omit the metadata operations and focus on the impact of cleaning overhead of LFS. This omission greatly simplified the design and implementation of the disk layout simulator. Since LFS performs much better than Overwrite on metadata operations [19, 21], the omission of metadata operations makes our results for LFS, WOLF, and HyLog conservative compared to Overwrite.

These assumptions, however, do not mean that HyLog can only be used at the storage level with NVRAM. When technologies such as Soft-updates [4] and journaling [28] are applied to HyLog, it could be used at the file system level as well.

WOLF [26] reduces the segment cleaning cost of LFS by sorting the pages to be written based on their update frequencies and writing to multiple segments at a time. This idea can be easily applied to HyLog to further reduce its cleaning cost, but, to isolate the benefit realized from the design of HyLog and from the idea of WOLF, this optimization is not performed in this paper.

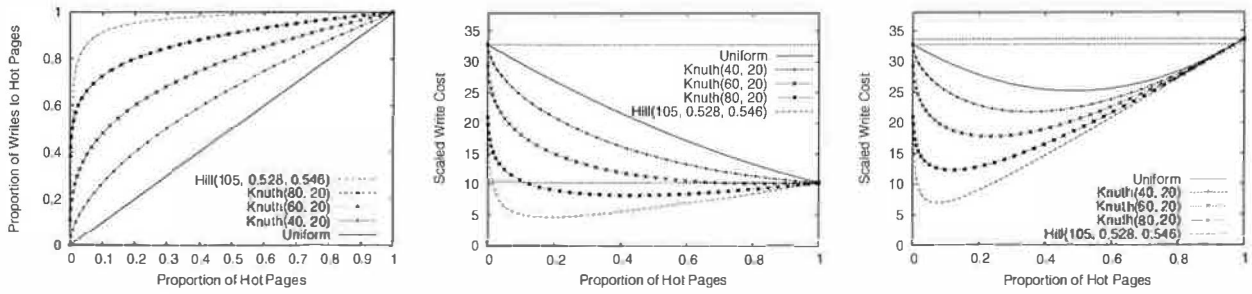
4.2 Separating Algorithm

Before a page is written to the disk, HyLog runs a *separating algorithm* to determine if this page is hot. If it is, the write is delayed and the page is stored temporarily in an in-memory segment buffer. Otherwise, it is immediately overwritten to its original place on the disk. When hot pages fill up the segment buffer, they are written out to a free disk segment, freeing the disk space occupied by their old copies.

As time goes on, some hot pages may become cold. These pages are written to the cold partition rather than to their current locations in the hot partition to avoid extra cleaning overhead. As some cold pages become hot and are written to the hot partition, free space may appear in the cold partition. To reclaim this free space more effectively, HyLog uses an adaptive cleaning algorithm to select segments with the highest cleaning benefit from both hot and cold partitions.

Accurately separating hot pages from cold pages is the key to the design of HyLog, as shown in Figure 4. The basic idea of the separating algorithm is simple. First, the write frequencies of recently updated pages are collected. These write frequencies are used to get the relationship between w and h. Then Equation (8) is used to calculate C'_{hylog} for all h. The hot page proportion h with the lowest C'_{hylog} is used as the expected hot page proportion.

Accurately measuring η is important for HyLog to make correct decisions. The service time of page I/O and segment I/O of each request is collected at the disk level. The average of the most recent 10,000 requests is used to compute η. Since a segment I/O always keeps all disks busy, while one page I/O only keeps one disk busy, page I/O is less efficient in disk arrays. If the proportion of the disk idle time is



(a) Cumulative Distribution Functions (b) Disk Space Utilization is 90% (c) Disk Space Utilization is 97%

Figure 4: Performance Potential of HyLog.

[The two horizontal lines in Figure 4(b) and 4(c) represent the write cost of Overwrite and LFS, respectively. η is 32.8, representing Cheetah X15 36LP with 1MB segment size and 8KB page size. $Knuth(a, b)$ means a% of the references go to b% of the pages.]

P_{idle}, η is adjusted to $\eta/(1 - P_{idle})$.

The write frequencies of all disk pages are collected in real time. A frequency counter is associated with each page. This counter is initialized to 0, and reset to 0 after every *measurement interval*. Whenever a page is written to the disk, its frequency counter is incremented. At the end of each measurement interval, all frequency counters are sorted in a descending order and stored in an array, which is used to calculate hot writes given the hot page proportion. The separating algorithm is invoked every measurement interval. After the expected hot page proportion is obtained, a page separating threshold can be determined so that all pages with write frequencies no less than the threshold are considered hot pages.

Preliminary experiments were conducted to study the sensitivity of system performance to the value of the measurement interval. When the measurement interval is smaller than 20 minutes, the throughput is not sensitive to the measurement interval. However, the throughput starts dropping with larger measurement intervals. Since the separating algorithm is invoked every measurement interval, 20 minutes is used as the measurement interval to reduce the separating algorithm overhead.

4.3 Segment Cleaning Algorithm

We adapted HyLog's segment cleaning algorithm from the adaptive cleaning algorithm [14], which dynamically selects between cost-age cleaning and hole-plugging based on their write cost.

In Hylog, the cleaner is invoked whenever the number of free segments is below a threshold (set to 10). In every cleaning pass, the cleaner processes up to 20MB of data. It first calculates the cost-benefit values of the following four possible cleaning choices: (1) cost-age in the hot partition, (2) hole-plugging in the hot partition, (3) cost-age in the cold partition, and (4) hole-plugging in the cold partition. It then performs cleaning using the scheme with the lowest cost-benefit value.

5 Methodology

5.1 The Simulator, Verification, and Validation

We used trace-driven simulations to compare the throughput of different disk layouts. Our simulator consists of a disk component, a disk layout component, and a buffer pool component. We ported the disk component from DiskSim 2.0 [5]. The disk layout component simulates disk layouts for Overwrite, LFS, WOLF, and HyLog. The implementation of LFS is based on the description in [14, 19] and the source code of the Sprite operating system [23]. The implementation of WOLF is based on the description in [26]. The buffer pool component uses the LRU algorithm. The three components communicate through an event-driven mechanism. Overwrite, LFS and WOLF are implemented as special cases of HyLog. By considering all pages as cold, we get Overwrite, and, by treating all pages as hot, we get LFS and WOLF. Therefore, the only difference between these disk layouts is the page separating algorithm. This guarantees the fairness of the performance comparison. We validated the simulator carefully in this subsection to improve the credibility of our performance comparisons.

In order to verify the disk layout component, a simple disk layout simulator called *TinySim* was developed independently. TinySim simulates LFS and WOLF, and supports single user and single disk. TinySim and the disk layout component were

run under uniformly distributed random update and hot-cold (10% of the pages are referenced 90% of the time) synthetic workloads, respectively. The overall write cost, which is the key performance metric of LFS and WOLF [14, 26], was obtained from both simulators. In most cases, the differences between the results of the two simulators were within 5%.

After verification, the cleaning algorithms in the disk layout component were validated against those discussed in [14]. Figure 5 shows the overall write costs of the cost-age, hole-plugging, and adaptive cleaning algorithms under a uniformly distributed random update workload. These cleaning algorithms show trends very similar to those in Figure 6 of [14].

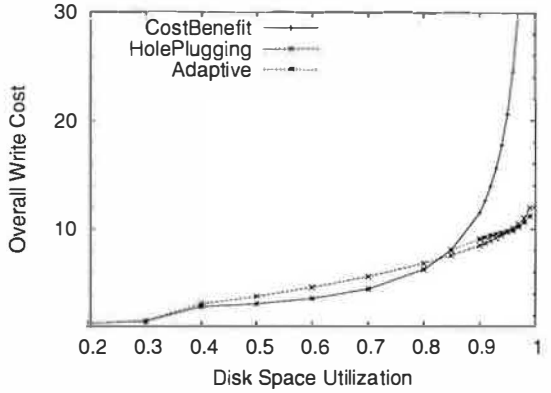


Figure 5: Validation of Overall Write Cost of LFS. [The workload is uniform random update. Page size is 8KB, segment size is 256KB, $T_{pos} = 15ms$, $B = 5MB/s$, and cache size is 128MB.]

We further validated our implementations of Overwrite and LFS by comparing their performance with that of FFS (uses overwrite) and BSD LFS published in [21] under the TPC-B benchmark. As in [14], we used the uniformly distributed random update workload to simulate the TPC-B benchmark. Since the DSP 3105 disk used in [21] is not available in DiskSim, a similar disk, the DEC RZ26, was used in the validation experiments. Table 2 lists the specifications of the two disks. The DEC RZ26 has 3% slower average seek time and slightly higher transfer rate because it has one more sector per track than the DSP 3105.

Table 3 shows the throughput of Overwrite and LFS obtained from our simulator and that in [21]. Although the reported throughput of LFS with cleaning in [21] was 27.0, it has been argued [17] that 34.4 should be a more reasonable value. Therefore, 34.4 is used here when calculating the difference.

Table 2: Disk Comparison for Simulator Validation

Parameters	DSP 3105	DEC RZ26
RPM	5400	5400
Sectors/Track	57	58
Cylinders	2568	2599
Platters	14	14
Track Buffer	256KB	285KB
Avg. Seek Time	9.5ms	9.8ms
Transfer Rate	2.3MB/s	2.3MB/s

The 4.8% lower throughput we observed for Overwrite in our experiments may be due to the 3.2% slower seek time of the DEC RZ26. The LFS implementation used in [21] can achieve only 1.7MB/s write throughput, 26% slower than the maximum hardware bandwidth, because of "missing a rotation between every 64 KB transfer". Since the number of segment reads is equal to the number of segment writes (for every segment read, there is always u segment write for cleaning and $1 - u$ segment write for new data), this slowdown of segment write should cause 13% performance difference, which matches the difference in Table 3. Since the differences in all results are within a reasonable range, we believe that our implementations of Overwrite and LFS are valid.

Table 3: Throughput Validation. $u_d = 50\%$.

Layout	Previous[21]	Ours	Diff.
FFS/Overwrite	27.0	25.7	-4.8%
LFS w/o cleaning	41.0	43.3	5.6%
LFS w cleaning	**27.0 (34.4)**	39.0	13.4%

5.2 The Workloads

We used three traces in our experiments: TPC-C™, Financial, and Campus. Their characteristics are summarized in Table 4.

The TPC-C benchmark is a widely accepted benchmark for testing the performance of database systems running OLTP workloads developed by the Transaction Processing performance Council (TPC) [24]. The TPC-C trace contains all read and write requests to the buffer pool when running the benchmark on IBM® DB2® 7.2 on Windows NT® Server 4.0. The scale of the TPC-C benchmark is expressed in terms of the number of warehouses represented. The database used in this study contains 50 warehouses. The TPC-C trace was collected using a tracing package ported from [8].

The Financial trace [22] was published by the Storage Performance Council. It is a disk I/O trace

Table 4: Trace Characteristics

	TPC-C	Financial	Campus
Data size(MB)	5088	10645	9416
Page size(KB)	4	4	8
#reads($\times 10^6$)	176.27	0.97	21.05
#writes($\times 10^6$)	31.17	3.41	7.64
Logical reads/writes	5.66	0.28	2.76
Physical reads/writes	1.37	0.13	2.56

of an OLTP application running at a large financial institution. Since the trace was collected at the I/O controller level, many reads have already been filtered out by the in-memory buffer. This trace contains I/O requests to 23 containers. In our experiments, the requests to the three largest containers were ignored to reduce the resources required by the simulator. Since these three containers have the fewest requests relative to their sizes, this omission has little impact on our results.

The Campus trace is a one-day trace taken from the NFS trace collected at Harvard University campus in October 2001 [3]. This trace is dominated by email activities. It contains reads, writes and directory operations to the NFS server. Reads and writes make up 85% of the requests. Since the sizes of the directories are unknown from the trace, it is difficult to replay the directory operations in the simulator, but because we assume NVRAM is used, these directory operations do not cause expensive synchronized writes. So their impact on performance is small. We discard the directory operations and use only the reads and writes.

5.3 Experimental Setup

Since our interest is the performance of various disk layouts on busy systems, we configured the simulator as a closed system without think time, i.e., the next trace record is issued as soon as the processing of the previous one finishes. Using this method, we were able to use traces with a small number of users to represent the workloads imposed on a system by many more users with think time. For example, the workload generated by 30 users without think time with 1.28 seconds average response time is equivalent to that generated by about 500 users with 21 seconds[1] average think time between requests. The details of this deduction are in the appendix.

We used the Quantum atlas10k 1999 disk model, the latest disk model provided by DiskSim. Its specifications are given in Table 5. Write-back caching is disabled to protect data loss from power failure. The disk scheduling algorithm is SCAN based on logical page numbers.

[1]The weighted average think time plus keying time defined in Clause 5.2.5.7 of TPC-C benchmark version 5.0

Table 5: Disk Specifications

Parameters	Atlas10k (Year 1999)	Year 2003 Disk	Year 2008 Disk
RPM	10025	15000	24000
Sectors/Track	229-334	476	967
Cylinders	10042	10042	10042
Platters	6	8	8
Size(GB)	9.1	18	36
Seek Time(ms)	5.6	3.6	2.0
Bandwidth(MB/s)	20.4	61	198

To study the performance of disk layouts on today's and future disks, we also designed models for a high-end disk of year 2003 and a high-end disk of the sort we might imagine to appear in year 2008. Looking back over 15 years history of disk technology evolution, we made the following assumptions: every 5 years, transfer rate increases by 242% [6], average seek time decreases by 76% [20], and RPM (Rotations Per Minute) increases by 61% [1]. We also assumed that all cylinders have the same number of tracks, the number of platters is 8, and the disk size is 18GB for the year 2003 disk and 36GB for the year 2008 disk. The specifications of these two disks calculated on the basis of these assumptions are given in the two rightmost columns of Table 5. The seek time distribution data were created by linearly scaling the seek time distribution of the atlas10k disk defined in DiskSim.

We used RAID-0 and RAID-5 as multi-disk models. The stripe size for both RAID-0 and RAID-5 is computed based on Equation (6) and then rounded to the closest powers of two. For RAID-0 arrays with n disks, the segment size is $n \times StripeSize$. For RAID-5 arrays, the segment size is $(n-1) \times StripeSize$, since $1/n$ of the total disk space is dedicated to parity data.

In order to vary the disk space utilization, only part of the disk is accessed, independent of the actual size of the disk. For example, if the data size is 6GB and the disk space utilization is 60%, the total disk space required is $\frac{6GB}{60\%} = 10GB$. If there are 5 disks, the first 2GB of each disk is used. Since the disk layout approaches do not handle page allocation and deallocation, all data are stored on the former part of the disk initially. As a result, the seek time (particularly for Overwrite) is very short,

which makes η smaller. Thus this data layout makes the performance results for LFS, WOLF, and HyLog conservative compared to Overwrite than in real workloads where data are often placed far apart. For LFS, WOLF, and HyLog, the data will eventually spread across the whole disk as data are written, which is considered as the warmup period. Only the performance data collected after the warmup period is considered.

The performance metric used in this paper is throughput, defined as the number of I/O requests finished per second.

Table 6 summarizes the parameters and values used in our experiments. Since these parameters can be easily controlled in the TPC-C trace, this trace is used to study the impacts of various parameters on throughput. When evaluating the throughput of RAID-5, we compare a 9-disk RAID-5 array with an 8-disk RAID-0 array so that they have the same segment size.

Table 6: Experimental Parameters

Configuration	Range	Default
Disk layout	Overwrite, LFS, WOLF, HyLog	—
Number of users	1–30	20
Number of disks	1–15	4
Disk utilization	0.5–0.98	—
Disk type	atlas10k, year 2008 disk, year 2003 disk	atlas10k
Disk array type	RAID-0, RAID-5	RAID-0
Workload	TPC-C, Financial, Campus	TPC-C
Buffer pool size	—	400MB

6 Simulation Results

6.1 Validation of the Cost Model

Since the cost model was developed for uniform random update workload, we use results for the same workload to validate the cost model. In particular, we use previous results for TPC-B (reported in [17, 21]), a random update workload, to verify the throughput of LFS and Overwrite. Since the write cost is the average time required to write a page and a transaction requires a page read and a page write, the throughput X is computed as

$$X = \frac{1}{T_{pg} + C + T_{cpu}},$$

where C is the write cost of the disk layout, and T_{cpu} is the CPU overhead for processing each page, which is $0.9ms$ for Overwrite and $1.8ms$ for LFS [21]. The results in Table 7 show that the model matches the measurement results well.

Table 7: Cost Model Validation

Layout	Previous	Model	Difference
Overwrite	27.0 [21]	28.6	6.0%
LFS-cleaning	34.4 [17]	37.3	8.4%

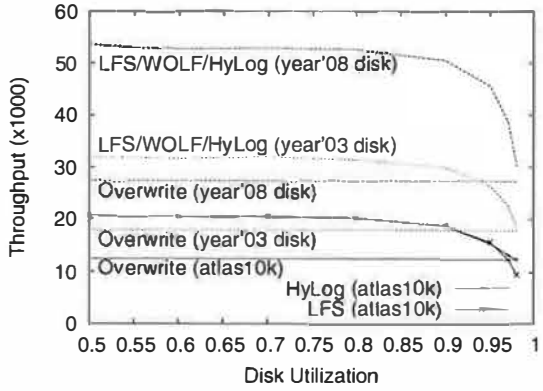
6.2 Impact of Various Factors

Disk Space Utilization and Disk Type

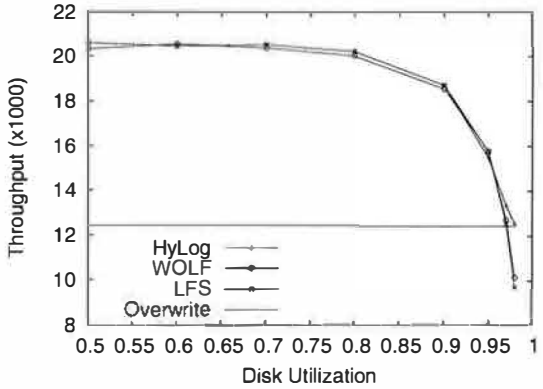
Figure 6(a) shows the throughput of different layouts under various disk space utilization and different disks. Since the throughput of LFS, WOLF, and HyLog almost overlaps for the year 2003 and year 2008 disks, only one line is shown for each of these disks. The throughput of all layouts improves with faster disks. The throughput of Overwrite is not affected by the disk space utilization, while the throughput of other layout approaches decreases when the space utilization is high. The faster the disk, the more LFS and WOLF can tolerate the high space utilization because faster disks have higher η as shown in Figure 1. Figure 6(b) gives a closer look at the throughput of the atlas10k disk. The throughput of WOLF overlaps that of LFS for most configurations and outperforms LFS by 5% when the disk space utilization u_d is very high (98%). The throughput of HyLog overlaps that of LFS when $u_d \leq 95\%$. This is because HyLog considers all pages as hot based on its cost model Equation (8) (see Figure 4(b)). The throughput of HyLog is comparable with Overwrite when the disk space utilization is higher. HyLog outperforms Overwrite by 7.4% when the disk space utilization is 97%.

To provide some insights into the performance that LFS and HyLog show above, we give further analysis of two example points: LFS running on atlas10k with 95% disk space utilization and HyLog running on atlas10k with 98% disk space utilization.

In the LFS example, the measured cleaning space utilization is $u = 88.4\%$. This is lower than the 90.2% computed from Equation (2) because of the skewness of accesses in TPC-C. Therefore, to write one segment of new data, $\frac{1+u}{1-u} = 16.3$ segment I/Os need to be performed for cleaning. So the cleaning traffic contributes 94.2% to the total segment I/O traffic. The measured T_{pg} and T_{seg} from simulation is 5.6ms and 27.3ms, respectively. Therefore, $\eta = 26.3$. The measured proportion of disk idle time is



(a) All Disks



(b) Atlas10k Disk

Figure 6: The Impact of Disk Space Utilization on System Throughput.

[The throughput is normalized to Overwrite. The number of users is 20, the number of disks is 4, the trace is TPC-C, and the buffer pool size is 400MB.]

30%, so η should be adjusted to $\eta/(1-30\%) = 37.6$. Based on the scaled write cost model,

$$C'_{ow}/C'_{lfscleaning} = \eta(1-u)/2 = 2.2,$$

which means the write throughput of LFS is 2.2 times of the write throughput of Overwrite. Since the write traffic contributes 42% to the total traffic after filtering by the buffer cache (Table 4), LFS outperforms Overwrite by 30%, which is close to the simulation result of 27%.

In the HyLog example, the hot page proportion selected by the page separating algorithm during the run is 35-45%. We use the data collected at the first measurement interval after warmup as the example. The proportion of hot pages is 42.2%, and the proportion of hot writes is 58.2%. The measured cleaning space utilization is 93.4%, which is lower than that in LFS for the same configuration (96.2%). The proportion of disk idle time is 22.5%, the measured T_{pg} and T_{seg} are 5.8ms and 27.2ms, respectively, and the adjusted η is $\frac{T_{pg}S}{T_{seg}(1-P_{idle})} = 35.2$. Therefore, the write cost model indicates that the write throughput of the hot partition is 16% higher than Overwrite. Thus the overall weighted write throughput is 9% higher than Overwrite. Taking the read traffic into account, the throughput of HyLog is 1.036 that of Overwrite, which is close to the simulation result of 1.008. The write throughput of LFS computed from the cost model under 98% disk space utilization is 66.9% of Overwrite, and the overall throughput of LFS including read and write traffic is 82.8% of Overwrite, which is close to the simulation result of 78.0%.

Figure 7 shows how well the separating algorithm works. The hot page proportion found by the separating algorithm (35-45%) is close to optimal, and the achieved throughput is 96.4% of the maximum possible throughput.

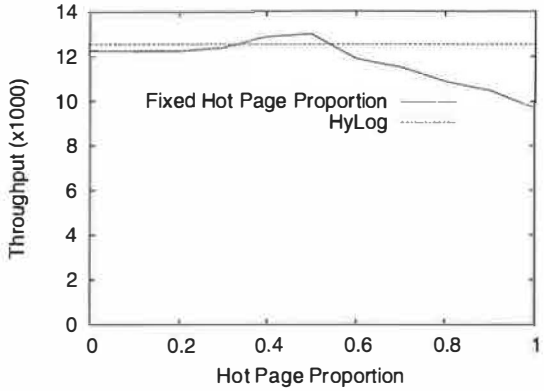


Figure 7: Sensitivity of Separating Criteria.

[The number of users is 20, the number of disks is 4, the trace is TPC-C, the disk space utilization is 98%, and the buffer pool size is 400MB.]

Impact of Number of Users and Number of Disks

Figure 8 shows the throughput normalized to Overwrite under different numbers of users and disks. WOLF is not shown since it almost overlaps with LFS. Two trends can be observed in the relative throughput of LFS, WOLF, and HyLog: (1) it drops with more users; (2) it drops with more disks.

With more users, the average disk seek time is reduced because of the disk scheduling algorithm, which reduces η. The disk idle time in Overwrite is also reduced with more users. Therefore, the first trend happens in both low disk space utilization

(Figure 8(a)) and high disk space utilization (Figure 8(b) and 8(c)).

With more disks, the segment size is larger, thus the cleaning cost is higher [14], which reduces the benefit of the log-structured layout. This happens only when cleaning cost plays an important role, which is true when the disk space utilization is high. Therefore, the second trend is apparent only when the disk space utilization is high (Figure 8(b)).

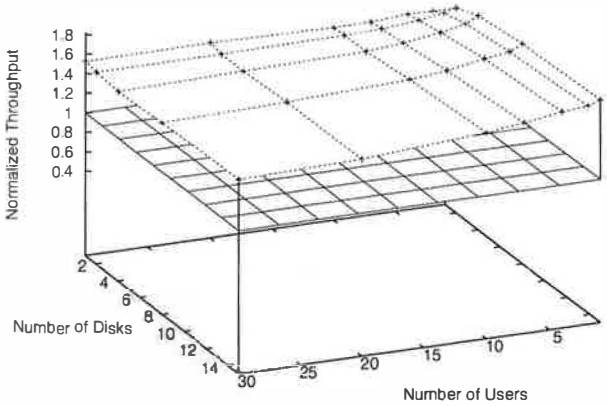
In Figure 8(b) and 8(c), the throughput of HyLog overlaps with that of LFS when LFS outperforms Overwrite, and HyLog becomes comparable with Overwrite when Overwrite outperforms LFS. HyLog incorrectly follows LFS when there are 4 users and 15 disks, because at this configuration, a very small error in the estimation of η can cause HyLog to make the wrong decision, while HyLog can tolerate some error in the estimation of η in other configurations.

Impact of Disk Array Type

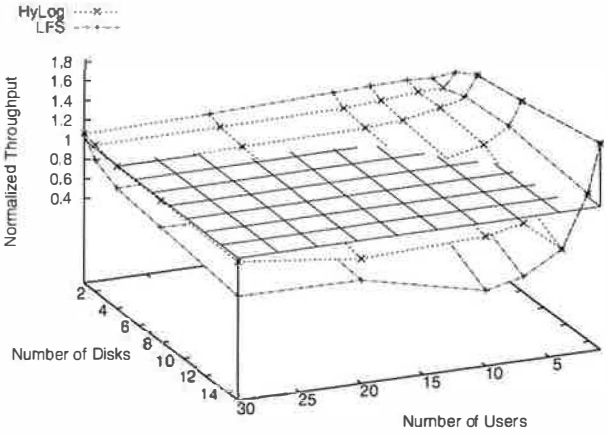
Figure 9 shows the throughput of the four disk layouts (Overwrite, LFS, WOLF, and HyLog) on RAID-0 and RAID-5. For Overwrite, the throughput on RAID-5 is about 50% of that on RAID-0. This performance degradation is caused by the slower page update of RAID-5. For LFS and WOLF, the use of RAID-5 increases throughput by 6.5-10%, because the segment I/O performance is not affected by RAID-5, while the one more disk in RAID-5 increases the page read throughput. When the disk space utilization is high, the throughput of HyLog on RAID-0 is comparable with Overwrite. The throughput of HyLog on RAID-5 is comparable with LFS because the slower page I/O in RAID-5 makes η higher. Thus most pages are considered to be hot pages.

6.3 Results for the Other Workloads

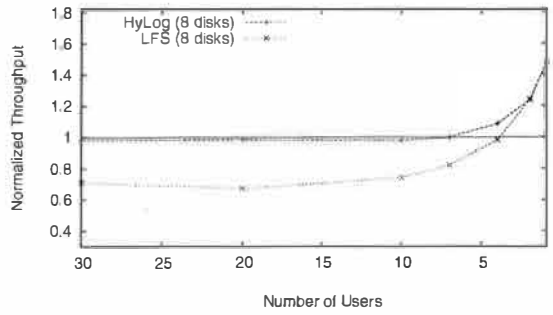
Figure 10 shows the throughput of the four disk layouts using the Financial and Campus traces. The throughput is normalized relative to that of Overwrite. For both traces, the performance advantage of LFS, WOLF, and HyLog is much higher than that observed with the TPC-C trace. This difference is attributed to two facts. First, the distribution of data updates in the Financial and Campus traces is more skewed than it is in the TPC-C trace, leading to less cleaning cost. Second, the proportion of writes in these two traces is much higher than in the TPC-C trace, since many reads have already been filtered out by client-side buffers (in the Campus trace) or in-memory buffers (in the Financial trace). Since the Financial trace is more skewed than the Campus trace and the writes in the Financial trace have higher proportion than in the Campus trace, the advantage of log-structured layouts in the Financial trace is higher than in the Campus trace. The performance results under other configurations



(a) Disk Space Utilization 90%



(b) Disk Space Utilization 98%



(c) Disk Space Utilization 98%

Figure 8: Impact of Number of Users and Number of Disks.
[Disk is atlas10k, trace is TPC-C. In Figure 8(a), the throughput curves of LFS and HyLog almost overlap, thus only the throughput of LFS is drawn. Figure 8(c) shows the hidden data points of Figure 8(b).]

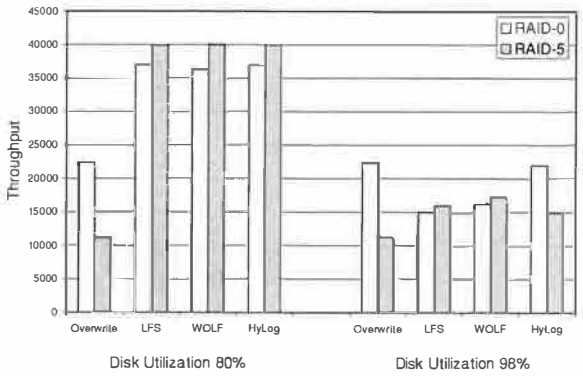


Figure 9: Throughput under RAID-0 and RAID-5 arrays.

[The RAID-0 array contains 8 disks, the RAID-5 array contains 9 disks, the disk is the atlas10k, the trace is TPC-C, the number of users is 20, the segment size is 512KB per disk, and the buffer pool size is 400MB.]

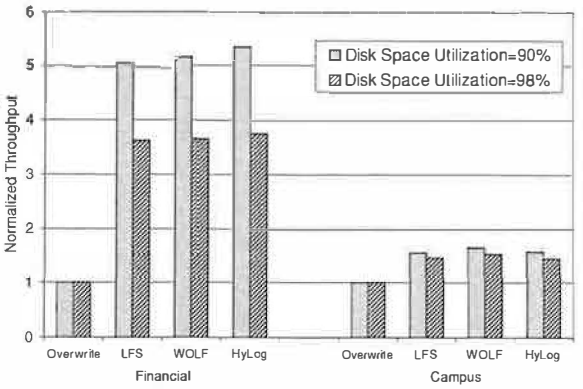


Figure 10: Throughput of LFS, WOLF, and HyLog normalized relative to Overwrite for the Financial and Campus traces.

[The number of disks is 1, the disk space utilization is 80%, the segment size is 512KB, and the buffer pool size is 400MB.]

show similar trends.

7 Conclusions and Future Work

In this paper we study the write performance of the Overwrite and LFS disk layouts. A write cost model was developed to compare the performance of these disk layouts. Contrary to the common belief that its high cleaning cost disadvantages LFS, we found that because of the advancement of disk technologies, the performance of LFS is significantly better than Overwrite even under the most pathological workload for LFS (random update), unless the disk space utilization is very high.

Since LFS still performs worse than Overwrite under certain conditions such as high disk space utilization, we propose a new disk layout model called HyLog. HyLog uses a log-structured approach for hot pages to achieve high write performance and overwrite for cold pages to reduce the cleaning cost. The page separating algorithm of HyLog is based on the write cost model and can separate hot pages from cold pages dynamically. Our results on a wide range of system and workload configurations show that HyLog performs comparably to the best of Overwrite, LFS, and WOLF in most configurations.

As future work, we want to study the read performance of LFS and HyLog and the impact of disk technology on them.

Acknowledgements

We would like to thank our shepherd Edward Lee and the anonymous reviewers for their insightful comments, which greatly improved the paper. We also thank Professor Derek Eager, Professor Dwight Makaroff, and Greg Oster for their feedback. Our system administrators, Greg Oster, Brian Gallaway, Cary Bernath, and Dave Bocking, provided the enormous computing resources for this study, and Daniel Ellard provided the Campus trace. This work was supported by IBM's Centre for Advanced Studies (CAS) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Appendix

The average think time plus keying time defined by the TPC-C benchmark is 21 seconds. The simulation results indicate that the system with 30 users without think time has a response time of 1.28 seconds if one disk is present in the system. Assuming that the number of users with think time in the system is n, the average arrival rate of users is $\frac{n}{21+1.28} = \frac{n}{22.28}$. From Little's Law, we have: $30 = \frac{n}{22.28} \times 1.28$. Therefore, $n = 522 \approx 500$, which indicates that the workload generated by 30 users without think time presents equivalent workload to that generated by about 500 users with 21 seconds think time between requests.

References

[1] D. Anderson, J. Dykes, and E. Riedel. More than an interface — SCSI vs. ATA. In *Proc. FAST'03*, pages 245–257, San Francisco, CA, March 2003.

[2] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proc. USENIX ATC'95*, pages 277–288, New Orleans, LA, January 1995.

[3] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proc. FAST'03*, pages 203–216, San Francisco, CA, March 2003.

[4] G. Ganger, M. McKusick, C. Soules, and Y. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Trans. on Computer Systems*, 18(2):127–153, 2000.

[5] G. Ganger, B. Worthington, and Y. Patt. *The DiskSim Simulation Environment Version 2.0 Reference Manual*, December 1999.

[6] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proc. 16th Intl. Conf. on Data Engineering*, pages 3–12, San Diego, CA, February 2000.

[7] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. USENIX Winter ATC'94*, pages 235–246, San Francisco, CA, January 1994.

[8] W. Hsu, A. Smith, and H. Young. Characteristics of production database workloads and the TPC benchmarks. *IBM Systems Journal*, 40(3):781–802, 2001.

[9] Y. Hu and Q. Yang. DCD – disk caching disk: A new approach for boosting I/O performance. In *Proc. Intl. Symp. on Computer Architecture*, pages 169–178, Philadelphia, PA, May 1996.

[10] L. Huang and T. Chiueh. Experiences in building a software-based SATF disk scheduler. Technical Report ECSL-TR81, State University of New York, Stony Brook, March 2000. revised in July, 2001.

[11] D. Knuth. *The Art of Computer Programming – Sorting and Searching*, volume 3. Addison-Wesley, second edition, 1998.

[12] D. Lomet. The case for log structuring in database systems. In *Proc. Intl. Workshop on High Performance Transaction Systems*, September 1995.

[13] C. Lumb, J. Schindler, and G. Ganger. Freeblock scheduling outside of disk firmware. In *Proc. FAST'02*, pages 275–288, Monterey, CA, January 2002.

[14] J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proc. SOSP'97*, pages 238–251, Saint-Malo, France, October 1997.

[15] J. Menon. A performance comparison of RAID-5 and log-structured arrays. In *IEEE Symp. on High-Performance Distributed Computing*, pages 167–178, Charlottesville, VI, August 1995.

[16] J. Menon and L. Stockmeyer. An age-threshold algorithm for garbage collection in log-structured arrays and file systems. IBM Research Report RJ 10120, IBM Research Division, San Jose, CA, 1998.

[17] J. Ousterhout. The second critique of Seltzer's LFS measurements. `http://www.eecs.harvard.edu/~margo/usenix.195/ouster_critique2.html`.

[18] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1):11–28, January 1989.

[19] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. on Computer Systems*, 10(1):26–52, 1992.

[20] J. Schindler, J. Griffin, C. Lumb, and G. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *Proc. FAST'02*, pages 259–274, Monterey, CA, January 2002.

[21] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proc. USENIX ATC'95*, pages 249–264, New Orleans, LA, January 1995.

[22] Storage Performance Council I/O traces. `http://traces.cs.umass.edu/storage/Financial1.spc`.

[23] The Sprite operating system. `http://www.cs.berkeley.edu/projects/sprite/sprite.html`.

[24] Transaction Processing Performance Council. `http://www.tpc.org/`.

[25] J. Wang and Y. Hu. PROFS–performance-oriented data reorganization for log-structured file systems on multi-zone disks. In *Proc. MASCOTS'01*, pages 285–292, Cincinnati, OH, August 2001.

[26] J. Wang and Y. Hu. WOLF – a novel reordering write buffer to boost the performance of log-structured file systems. In *Proc. FAST'02*, pages 46–60, Monterey, CA, January 2002.

[27] R. Wang, T. Anderson, and D. Patterson. Virtual log based file systems for a programmable disk. In *Proc. OSDI'99*, pages 29–43, New Orleans, LA, February 1999.

[28] Z. Zhang and K. Ghose. yFS: A journaling file system design for handling large data sets with reduced seeking. In *Proc. FAST'03*, pages 59–72, San Francisco, CA, March 2003.

Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks

Jiri Schindler*, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, Gregory R. Ganger
Carnegie Mellon University

Abstract

The *Atropos* logical volume manager allows applications to exploit characteristics of its underlying collection of disks. It stripes data in track-sized units and explicitly exposes the boundaries, allowing applications to maximize efficiency for sequential access patterns even when they share the array. Further, it supports efficient diagonal access to blocks on adjacent tracks, allowing applications to orchestrate the layout and access to two-dimensional data structures, such as relational database tables, to maximize performance for both row-based and column-based accesses.

1 Introduction

Many storage-intensive applications, most notably database systems and scientific computations, have some control over their access patterns. Wanting the best performance possible, they choose the data layout and access patterns they believe will maximize I/O efficiency. Currently, however, their decisions are based on manual tuning knobs and crude rules of thumb. Application writers know that large I/Os and sequential patterns are best, but are otherwise disconnected from the underlying reality. The result is often unnecessary complexity and inefficiency on both sides of the interface.

Today's storage interfaces (e.g., SCSI and ATA) hide almost everything about underlying components, forcing applications that want top performance to guess and assume [7, 8]. Of course, arguing to expose more information highlights a tension between the amount of information exposed and the added complexity in the interface and implementations. The current storage interface, however, has remained relatively unchanged for 15 years, despite the shift from (relatively) simple disk drives to large disk array systems with logical volume managers (LVMs). The same information gap exists inside disk array systems—although their LVMs sit below a host's storage interface, most do not exploit device-specific features of their component disks.

This paper describes a logical volume manager, called *Atropos* (see Figure 1), that exploits information about its component disks and exposes high-level information about its data organization. With a new data organization and minor extensions to today's storage interface, it accomplishes two significant ends. First, *Atropos* exploits automatically-extracted knowledge of disk track boundaries, using them as its stripe unit boundaries. By also exposing these boundaries explicitly, it allows applications to use previously proposed "track-aligned extents" (*traxtents*), which provide substantial benefits for mid-sized segments of blocks and for streaming patterns interleaved with other I/O activity [22].

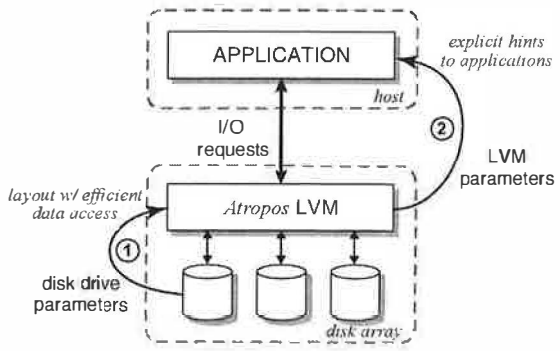


Figure 1: ***Atropos* logical volume manager architecture.** *Atropos* exploits disk characteristics (arrow 1), automatically extracted from disk drives, to construct a new data organization. It exposes high-level parameters that allow applications to directly take advantage of this data organization for efficient access to one- or two-dimensional data structures (arrow 2).

Second, *Atropos* uses and exposes a data organization that lets applications go beyond the "only one dimension can be efficient" assumption associated with today's linear storage address space. In particular, two-dimensional data structures (e.g., database tables) can be laid out for almost maximally efficient access in both row- and column-orders, eliminating a trade-off [15] currently faced by database storage managers. *Atropos* enables this by exploiting automatically-extracted knowledge of track/head switch delays to support *semi-sequential* access: diagonal access to ranges of blocks (one range per track) across a sequence of tracks.

In this manner, a relational database table can be laid out such that scanning a single column occurs at streaming bandwidth (for the full array of disks), and reading a single row costs only 16%–38% more than if it had been the optimized order. We have implemented *Atropos* as a host-based LVM, and we evaluate it with both database workload experiments (TPC-H) and analytic models. Because *Atropos* exposes its key parameters explicitly, these performance benefits can be realized with no manual tuning of storage-related application knobs.

*Now with EMC Corporation.

The rest of the paper is organized as follows. Section 2 discusses current logical volume managers and the need for *Atropos*. Section 3 describes the design an implementation of *Atropos*. Section 4 describes how *Atropos* is used by a database storage manager. Section 5 evaluates *Atropos* and its value for database storage management. Section 6 discusses related work.

2 Background

This section overviews the design of current disk array LVMs, which do not exploit the performance benefits of disk-specific characteristics. It highlights the features of the *Atropos* LVM, which addresses shortcomings of current LVMs, and describes how *Atropos* supports efficient access in both column- and row-major orders to applications accessing two-dimensional data structures.

2.1 Conventional LVM design

Current disk array LVMs do not sufficiently exploit or expose the unique performance characteristics of their individual disk drives. Since an LVM sits below the host's storage interface, it could internally exploit disk-specific features without the host being aware beyond possibly improved performance. Instead, most use data distribution schemes designed and configured independently of the underlying devices. Many stripe data across their disks, assigning fixed-sized sets of blocks to their disks in a round-robin fashion; others use more dynamic assignment schemes for their fixed-size units. With a well-chosen unit size, disk striping can provide effective load balancing of small I/Os and parallel transfers for large I/Os [13, 16, 18].

A typical choice for the stripe unit size is 32–64 KB. For example, EMC's Symmetrix 8000 spreads and replicates 32 KB chunks across disks [10]. HP's AutoRAID [25] spreads 64 KB "relocation blocks" across disks. These values conform to the conclusions of early studies [4] of stripe unit size trade-offs, which showed that a unit size roughly matching a single disk track (32–64 KB at the times of these systems' first implementations) was a good rule-of-thumb. Interestingly, many such systems seem not to track the growing track size over time (200–350 KB for 2002 disks), perhaps because the values are hard-coded into the design. As a consequence, medium- to large-sized requests to the array result in suboptimal performance due to small inefficient disk accesses.

2.2 Exploiting disk characteristics

Track-sized stripe units: *Atropos* matches the stripe unit size to the exact track size of the disks in the volume. In addition to conforming to the rule-of-thumb as disk technology progresses, this choice allows applications (and the array itself [11]) to utilize track-based accesses: accesses aligned and sized for one track. Recent research [22] has shown that doing so increases disk efficiency by up to 50% for streaming applications that share the disk system with other activity and for components (e.g., log-structured file systems [17]) that utilize medium-sized segments. In fact, track-based access provides almost the same disk efficiency for such applications as would sequential streaming.

The improvement results from two disk-level details. First, firmware support for zero-latency access eliminates rotational latency for full-track accesses; the data of one track can be read in one revolution regardless of the initial rotational offset after the seek. Second, no head switch is involved in reading a single track. Combined, these positioning delays represent over a third of the total service time for non-aligned, approximately track-sized accesses. Using small stripe unit sizes, as do the array controllers mentioned above, increases the proportion of time spent on these overheads.

Atropos uses automated extraction methods described in previous work [21, 22] to match stripe units to disk track boundaries. As detailed in Section 3.3, *Atropos* also deals with multi-zoned disk geometries, whereby tracks at different radial distances have different numbers of sectors, that are not multiples of any useful block size.

Efficient access to non-contiguous blocks: In addition to exploiting disk-specific information to determine its stripe unit size, *Atropos* exploits disk-specific information to support efficient access to data across several stripe units mapped to the same disk. This access pattern, called *semi-sequential*, reads some data from each of several tracks such that, after the initial seek, no positioning delays other than track switches are incurred. Such access is appropriate for two-dimensional data structures, allowing efficient access in both row- and column-major order.

In order to arrange data for efficient semi-sequential access, *Atropos* must know the track switch time as well as the track sizes. Carefully deciding how much data to access on each track, before moving to the next, allows *Atropos* to access data from several tracks in one full revolution by taking advantage of the Shortest-Positioning-Time-First (SPTF) [12, 23] request scheduler built into disk firmware. Given the set of accesses to the different tracks, the scheduler can arrange them to ensure efficient execution by minimizing the total positioning time. If the sum of the data transfer times and the track switch times equals the time for one rotation, the scheduler will service them in an order that largely eliminates rotational latency (similar to the zero-latency feature for single track access). The result is that semi-sequential accesses are much more efficient than a like number of random or unorchestrated accesses.

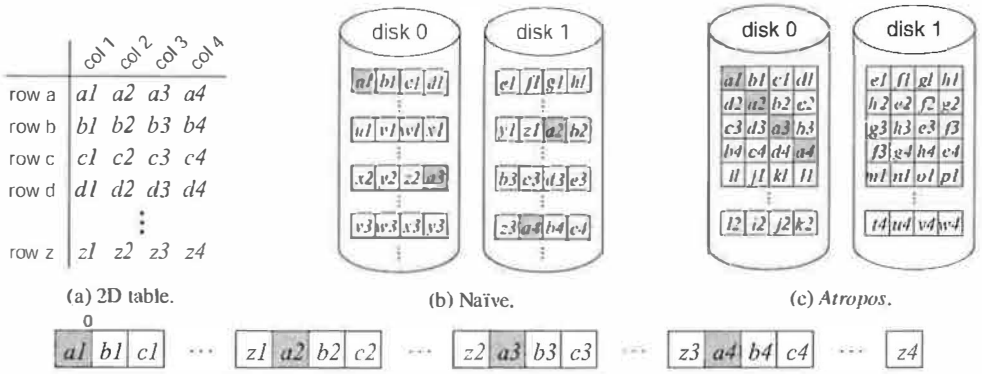


Figure 2: **Two layouts for parallel access to a two-dimensional data structure mapped to a linear *LBN* address space.**

2.3 Access to 2D data structures

Figure 2 uses a simple example to illustrate *Atropos*'s benefits to applications that require efficient access to two-dimensional structures in both dimensions and contrasts it with conventional striping in disk arrays. The example depicts a two-dimensional data structure (e.g., a table of a relational database) consisting of four columns $[1,\ldots,4]$ and many rows $[a,\ldots,z]$. For simplicity, each element (e.g., a_1) maps to a single *LBN* of the logical volume spanning two disks.

To map this two-dimensional structure into a linear space of *LBN*s, conventional systems decide a priori which order (i.e., column- or row-major) is likely to be accessed most frequently [5]. In the example in Figure 2, a column-major access was chosen and hence the runs of $[a_1,b_1,\ldots,z_1]$, $[a_2,b_2,\ldots,z_2]$, $[a_3,b_3,\ldots,z_3]$, and $[a_4,b_4,\ldots,z_4]$ assigned to contiguous *LBN*s. The mapping of each element to the *LBN*s of the individual disks is depicted in Figure 2(b) in a layout called *Naïve*. When accessing a column, the disk array uses (i) sequential access within each disk and (ii) parallel access to both disks, resulting in maximum efficiency.

Accessing data in the other order (i.e., row-major), however, results in disk I/Os to disjoint *LBN*s. For the example in Figure 2(b), an access to row $[a_1,a_2,a_3,a_4]$ requires four I/Os, each of which includes the high positioning cost for a small random request. The inefficiency of this access pattern stems from the lack of information in conventional systems; one column is blindly allocated after another within the *LBN* address space.

Atropos supports efficient access in both orders with a new data organization, depicted in Figure 2(c). This layout maps columns such that their respective first row elements start on the same disk and enable efficient row-order access. This layout still achieves sequential, and hence efficient, column-major access, just like the *Naïve* layout. Accessing the row $[a_1,a_2,a_3,a_4]$, however, is much more efficient than with *Naïve*. Instead of small random accesses, the row is now accessed semi-sequentially in (at most) one disk revolution, incurring much smaller positioning cost (i.e., eliminating all but the first seek and *all* rotational latency). Section 3 describes why this semi-sequential access is efficient.

2.4 Efficient access for database systems

By mapping two-dimensional structures (e.g., large non-sparse matrices or database tables) into a linear *LBN* space without providing additional information to applications, efficient accesses in conventional storage systems are only possible in one of row- or column-major order. Database management systems (DBMS) thus predict the common order of access by a workload and choose a layout optimized for that order, knowing that accesses along the other major axis will be inefficient.

In particular, online transaction processing (OLTP) workloads, which make updates to full records, favor efficient row-order access. On the other hand, decision support system (DSS) workloads often scan a subset of table columns and get better performance using an organization with efficient column-order access [15]. Without explicit support from the storage device, however, a DBMS system cannot efficiently support both workloads with one data organization.

The different storage models (a.k.a. page layouts) employed by DBMSs trade the performance of row-major and column-major order accesses. The page layout prevalent in commercial DBMS, called the N-ary storage model (NSM), stores a fixed number of full records (all n attributes) in a single page (typically 8 KB). This page layout is optimized for OLTP workloads with row-major access and random I/Os. This layout is also efficient for scans of entire tables; the DBMS can sequentially scan one page after another. However, when only a subset of attributes is desired (e.g., the column-major access prevalent in DSS workloads), the DBMS must fetch full pages with *all* attributes, effectively reading the entire table even though only a fraction of the data is needed.

To alleviate the inefficiency of column-major access with NSM, a decomposition storage model (DSM) vertically partitions a table into individual columns [5]. Each DSM page thus contains a *single* attribute for a fixed number of records. However, fetching full records requires n accesses to single-attribute pages and $n-1$ joins on the record ID to reconstruct the entire record.

The stark difference between row-major and column-major efficiencies for the two layouts described above is so detrimental to database performance that some have even proposed maintaining two copies of each table to avoid it [15]. This solution requires twice the capacity and must propagate updates to each copy to maintain consistency. With *Atropos*'s data layout, which offers efficient access in both dimensions, database systems do not have to compromise.

2.5 A more explicit storage interface

Virtually all of today's disk arrays use an interface (e.g., SCSI or ATA) that presents the storage device as a linear space of equally-sized blocks. Each block is uniquely addressed by an integer, called a logical block number (*LBN*). This linear abstraction hides non-linearities in storage device access times. Therefore, applications and storage devices use an unwritten contract, which states that large sequential accesses to contiguous *LBN*s are much more efficient than random accesses and small I/O sizes. Both entities work hard to abide by this implicit contract; applications construct access patterns that favor large I/O and LVMs map contiguous *LBN*s to media locations that ensure efficient execution of sequential I/Os. Unfortunately, an application decides on I/O sizes without any more specific information about the *LBN* mappings chosen by an LVM because current storage interfaces hide it.

In the absence of clearly defined mechanisms, applications rely on knobs that must be manually set by a system administrator. For example, the IBM DB2 relational database system uses the `PREFETCHSIZE` and `EXTENTSIZE` parameters to determine the maximal size of a prefetch I/O for sequential access and the number of pages to put into a single extent of contiguous *LBN*s [6]. Another parameter, called `DB2 STRIPED CONTAINERS`, instructs DBMS to align I/Os on stripe unit boundaries. Relying on proper knob settings is fragile and prone to human errors: it may be unclear how to relate them to LVM configuration parameters. Because of these difficulties, and the information gap introduced by inexpressive storage interfaces, applications cannot easily take advantage of significant performance characteristics of modern disk arrays. *Atropos* exposes explicit information about stripe unit sizes and semi-sequential access. This information allows applications to directly match their access patterns to the disk array's characteristics.

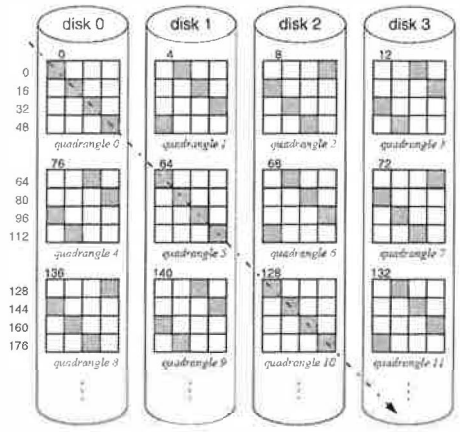


Figure 3: ***Atropos* quadrangle layout.** The numbers to the left of disk 0 are the *VLBN*s mapped to the gray disk locations connected by the arrow (not the first block of each quadrangle row). The arrow illustrates efficient access in the other-major.

3 Atropos logical volume manager

The *Atropos* disk array LVM addresses the aforementioned shortcomings of many current disk array LVM designs. It exploits disk-specific characteristics to construct a new data organization. It also exposes high-level features of this organization to higher-levels of the storage stack, allowing them to directly take advantage of key device-specific characteristics. This section details the new data organization and the information *Atropos* exposes to applications.

3.1 Atropos data organization

As illustrated in Figure 3, *Atropos* lays data across p disks in basic allocation units called quadrangles. A quadrangle is a collection of logical volume *LBN*s, here referred to as *VLBN*s, mapped to a single disk. Each successive quadrangle is mapped to a different disk.

A quadrangle consists of d consecutive disk tracks, with d referred to as the quadrangle's *depth*. Hence, a single quadrangle is mapped to a contiguous range of a single disk's logical blocks, here referred to as *DLBN*s. The *VLBN* and *DLBN* sizes may differ; a single *VLBN* consists of b *DLBN*s, with b being the block size of a single logical volume block. For example, an application may choose a *VLBN* size to match its allocation units (e.g., an 8 KB database block size), while a *DLBN* is typically 512 bytes.

Each quadrangle's dimensions are $w \times d$ logical blocks (*VLBN*s), where w is the quadrangle width and equals the number of *VLBN*s mapped to a single track. In Figure 3, both d and w are four. The relationship between the dimensions of a quadrangle and the mappings to individual logical blocks of a single disk are described in Section 3.2.2.

The goal of the *Atropos* data organization is to allow efficient access in two dimensions. Efficient access of

the primary dimension is achieved by striping contiguous *VLBN*s across quadrangles on all disks. Much like ordinary disk arrays, which map *LBN*s across individual stripe units, each quadrangle row contains a contiguous run of *VLBN*s covering a contiguous run of a single disk's *DLBN*s on a single track. Hence, sequential access naturally exploits the high efficiency of track-based access explained in Section 2.2. For example, in Figure 3, an access to 16 sequential blocks starting at *VLBN* 0, will be broken into four disk I/Os executing in parallel and fetching full tracks: *VLBN*s 0–3 from disk 0, *VLBN*s 4–7 from disk 1, *VLBN*s 8–11 from disk 2, and *VLBN*s 12–15 from disk 3.

Efficient access to the secondary dimension is achieved by mapping it to semi-sequential *VLBN*s. Figure 3 indicates the semi-sequential *VLBN*s with a dashed line. Requests to the semi-sequential *VLBN*s in a single quadrangle are all issued together in a batch. The disk's internal scheduler then chooses the request that will incur the smallest positioning cost (the sum of seek and rotational latency) and services it first. Once the first request is serviced, servicing all other requests will incur only a track switch to the adjacent track. Thanks to the semi-sequential layout, no rotational latency is incurred for any of the subsequent requests, regardless of which request was serviced first.

Naturally, the sustained bandwidth of semi-sequential access is smaller than that of sequential access. However, semi-sequential access is more efficient than reading d effectively-random *VLBN*s spread across d tracks, as would be the case in a normal striped disk array. Accessing random *VLBN*s will incur rotational latency, averaging half a revolution per access. In the example of Figure 3, the semi-sequential access, depicted by the arrow, proceeds across *VLBN*s $0, 16, 32, \ldots, 240$ and occurs on all p disks, achieving the aggregate semi-sequential bandwidth of the disk array.

3.2 Quadrangle layout parameters

The values that determine efficient quadrangle layout depend on disk characteristics, which can be described by two parameters. The parameter N describes the number of sectors, or *DLBN*s, per track. The parameter H describes the track skew in the mapping of *DLBN*s to physical sectors. The layout and disk parameters are summarized in Table 1.

Track skew is a property of disk data layouts as a consequence of track switch time. When data is accessed sequentially on a disk beyond the end of a track, the disk must switch to the next track to continue accessing. Switching tracks takes some amount of time, during which no data can be accessed. While the track switch is in progress, the disk continues to spin, of course. Therefore, sequential *LBN*s on successive tracks are physically skewed so that when the switch is complete, the head will be positioned over the next sequential *LBN*. This skew is expressed as the parameter H which is the number of *DLBN*s that the head passes over during the track switch time.

Symbol	Name	Units
Quadrangle layout parameters		
p	Parallelism	# of disks
d	Quadrange depth	# of tracks
b	Block size	# of *DLBN*s
w	Quadrange width	# of *VLBN*s
Disk physical parameters		
N	Sectors per track	
H	Head switch	in *DLBN*s

Table 1: **Parameters used by Atropos.**

Figure 4 shows a sample quadrangle layout and its parameters. Figure 4(a) shows an example of how quadrangle *VLBN*s map to *DLBN*s. Along the x-axis, a quadrangle contains w *VLBN*s, each of size b *DLBN*s. In the example, one *VLBN* consists of two *DLBN*s, and hence $b = 2$. As illustrated in the example, a quadrangle does not always use all *DLBN*s when the number of sectors per track, N, is not divisible by b. In this case, there are R residual *DLBN*s that are not assigned to quadrangles. Figure 4(b) shows the physical locations of each b-sized *VLBN* on individual tracks, accounting for track skew, which equals 3 sectors (H= 3 *DLBN*s) in this example.

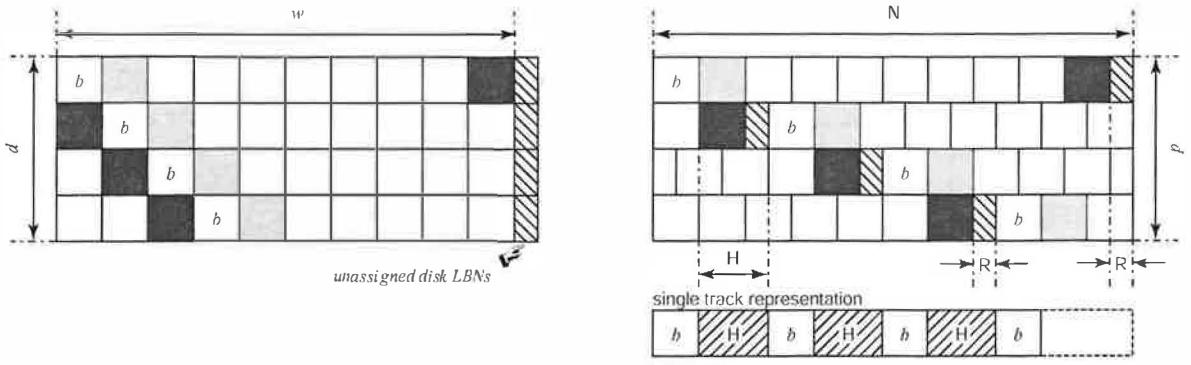
3.2.1 Determining layout parameters

To determine a suitable quadrangle layout at format time, *Atropos* uses as its input parameters the automatically extracted disk characteristics, N and H, and the block size, b, which are given by higher level software. Based on these input parameters, the other quadrangle layout parameters, d and w, are calculated as described below.

To explain the relationship between the quadrangle layout parameters and the disk physical parameters, let's assume that we want to read one block of b *DLBN*s from each of d tracks. This makes the total request size, S, equal to db. As illustrated in Figure 4(b), the locations of the b blocks on each track are chosen to ensure the most efficient access. Accessing b on the next track can commence as soon as the disk head finishes reading on the previous track and repositions itself above the new track. During the repositioning, H sectors pass under the heads.

To bound the response time for reading the S sectors, we need to find suitable values for b and d to ensure that the entire request, consisting of db sectors, is read in at most one revolution. Hence,

$$\frac{db}{N} + \frac{(d-1)H}{N} \leq 1 \qquad (1)$$



(a) **Logical layout in disk *LBN* space.** (b) **Physical layout on disk sectors.**

Figure 4: **Single quadrangle layout.** In this example, the quadrangle layout parameters are b=2 (a single *VLBN* consists of two *DLBN*s), w=10 *VLBN*s, and d= 4 tracks. The disk physical parameters are H=3 *DLBN*s and N=21 *DLBN*s. Given these parameters, R=1.

where db/N is the media access time needed to fetch the desired S sectors and $(d-1)H/N$ is the fraction of time spent in head switches when accessing all d tracks. Then, as illustrated at the bottom of Figure 4(b), reading db sectors is going to take the same amount of time as if we were reading $db+(d-1)H$ sectors on a single track of a zero-latency access disk.

The maximal number of tracks, d, from which at least one sector each can be read in a single revolution is bound by the number of head switches that can be done in a single revolution, so

$$d \leq \left\lfloor \frac{N}{H} \right\rfloor - 1 \tag{2}$$

If we fix d, the number of sectors, b, that yield the most efficient access (i.e., reading as many sectors on a single track as possible before switching to the next one) can be determined from Equation 1 to get

$$b \leq \frac{N+H}{d} - H \tag{3}$$

Alternatively, if we fix b, the maximal depth, called D_{max}, can be expressed from Equation 1 as

$$D_{max} \leq \frac{N+H}{b+H} \tag{4}$$

For certain values of N, db sectors do not span a full track. In that case, $db+(d-1)H < N$ and there are R residual sectors, where $R < b$, as illustrated in Figure 4. The number of residual *DLBN*s on each track not mapped to quadrangle blocks is $R = N \bmod w$, where

$$w = \left\lfloor \frac{N}{b} \right\rfloor \tag{5}$$

Hence, the fraction of disk space that is wasted with *Atropos*' quadrangle layout is R/N; these sectors are skipped to maintain the invariant that db sectors can be accessed in at most one revolution. Section 5.2.4 shows that this number is less than 2% of the total disk capacity.

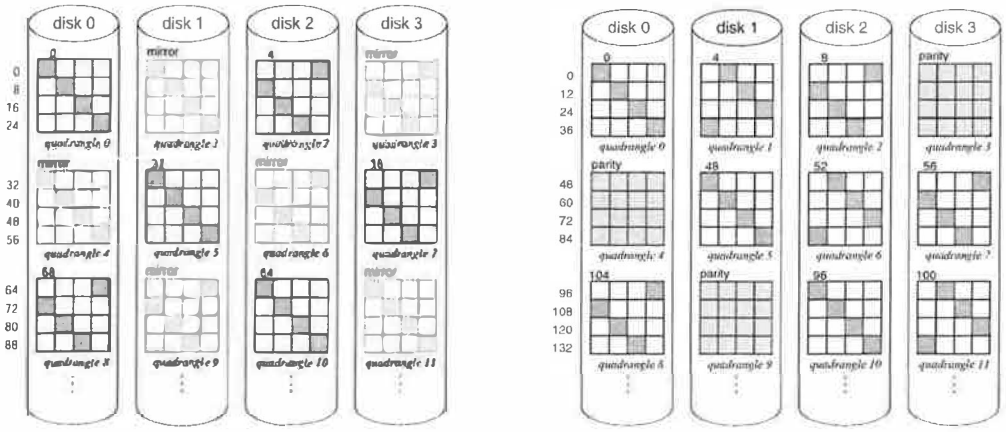
While it may seem that relaxing the one revolution constraint might achieve better efficiency, Appendix B shows that this intuition is wrong. Accessing more than D_{max} tracks is detrimental to the overall performance unless d is some multiple of D_{max}. In that case, the service time for such access is a multiple of one-revolution time.

3.2.2 Mapping *VLBN*s to quadrangles

Mapping *VLBN*s to the *DLBN*s of a single quadrangle is straightforward. Each quadrangle is identified by $DLBN_Q$, which is the lowest *DLBN* of the quadrangle and is located at the quadrangle's top-left corner. The *DLBN*s that can be accessed semi-sequentially are easily calculated from the N and b parameters. As illustrated in Figure 4, given $DLBN_Q = 0$ and $b = 2$, the set $\{0,24,48,72\}$ contains blocks that can be accessed semi-sequentially. To maintain rectangular appearance of the layout to an application, these *DLBN*s are mapped to *VLBN*s $\{0,10,20,30\}$ when $b = 2$, $p = 1$, and $VLBN_Q = DLBN_Q = 0$.

With no media defects, *Atropos* only needs to know the $DLBN_Q$ of the first quadrangle. The $DLBN_Q$ for all other quadrangles can be calculated from the N, d, and b parameters. With media defects handled via slipping (e.g., the primary defects that occurred during manufacturing), certain tracks may contain fewer *DLBN*s. If the number of such defects is less than R, that track can be used; if it is not, the *DLBN*s on that track must be skipped. If any tracks are skipped, the starting *DLBN* of each quadrangle row must be stored.

To avoid the overhead of keeping a table to remember the *DLBN*s for each quadrangle row, *Atropos* could reformat the disk and instruct it to skip over any tracks that contain one or more bad sectors. By examining twelve Seagate Cheetah 36ES disks, we found there were, on average, 404 defects per disk; eliminating all tracks with defects wastes less than 5% of the disk's total capacity. The techniques for handling grown defects still apply.



(a) RAID 1 layout.

(b) RAID 5 layout.

Figure 5: ***Atropos* quadrangle layout for different RAID levels.**

3.3 Practical system integration

Building an *Atropos* logical volume out of p disks is not difficult thanks to the regular geometry of each quadrangle. *Atropos* collects a set of disks with the same basic characteristics (e.g., the same make and model) and selects a disk zone with the desired number of sectors per track, N. The *VLBN* size, b, is set according to application needs, specifying the access granularity. For example, it may correspond to a file system block size or database page size. With b known, *Atropos* uses disk parameters to determine the resulting $d \leq D_{max}$.

In practice, volume configuration can be accomplished in a two-step process. First, higher-level software issues a FORMAT command with desired values of volume capacity, level of parallelism p, and block size b. Internally, *Atropos* selects appropriate disks (out of a pool of disks it manages), and formats the logical volume by implementing a suitable quadrangle layout.

3.3.1 Zoned disk geometries

With zoned-disk geometries, the number of sectors per track, N, changes across different zones, which affects both the quadrangle width, w, and depth, d. The latter changes because the ratio of N to H may be different for different zones; the track switch time does not change, but the number of sectors that rotate by in that time does. By using disks with the same geometries (e.g., same disk models), we opt for the simple approach: quadrangles with one w can be grouped into one logical volume and those with another w (e.g., quadrangles in a different zone) into a different logical volume. Since modern disks have fewer than 8 zones, the size of a logical volume stored across a few 72 GB disks would be tens of GBs.

3.3.2 Data protection

Data protection is an integral part of disk arrays and the quadrangle layout lends itself to the protection models of traditional RAID levels. Analogous to the parity unit, a set of quadrangles with data can be protected with a parity quadrangle. To create a RAID5 homologue of a parity group with quadrangles, there is one parity quadrangle unit for every $p-1$ quadrangle stripe units, which rotates through all disks. Similarly, the RAID 1 homologue can be also constructed, where each quadrangle has a mirror on a different disk. Both protection schemes are depicted in Figure 5.

3.3.3 Explicit information to applications

To allow applications to construct efficient streaming access patterns, *Atropos* needs to expose the parameter w, denoting the stripe unit size. I/Os aligned and sized to stripe unit boundaries can be executed most efficiently thanks to track-based access and rotating stripe units through all p disks. Applications with one-dimensional access (e.g., streaming media servers) then exercise access patterns consisting of w-sized I/Os that are aligned on disk track boundaries.

For applications that access two-dimensional data structures, and hence want to utilize semi-sequential access, *Atropos* also needs to expose the number of disks, p. Such applications then choose the primary order for data and allocate $w \times p$ blocks of this data, corresponding to a portion of column 1 $\{a_1, \ldots, h_1\}$ in Figure 2. They allocate to the next $w \times p$ *VLBN*s the corresponding data of the other-major order (e.g., the $\{a_2, \ldots, h_2\}$ portion of column 2) and so on, until all are mapped. Thus, the rectangular region $\{a_1, \ldots, h_4\}$ would be mapped to $4wp$ contiguous *VLBN*s.

Access in the primary-major order (columns in Figure 2) consists of sequentially reading wp *VLBN*s. Access in the other-major order is straightforward; the ap-

plication simply accesses every *wp*-th *VLBN* to get the data of the desired row. *Atropos* need not expose to applications the parameter *d*. It is computed and used internally by *Atropos*.

Because of the simplicity of information *Atropos* exposes to applications, the interface to *Atropos* can be readily implemented with small extensions to the commands already defined in the SCSI protocol. The parameters *p* and *w* could be exposed in a new mode page returned by the MODE SENSE SCSI command. To ensure that *Atropos* executes all requests to non-contiguous *VLBN*s for the other-major access together, an application can link the appropriate requests. To do so, the READ or WRITE commands for semi-sequential access are issued with the Link bit set.

3.3.4 Implementation details

Our *Atropos* logical volume manager implementation is a stand-alone process that accepts I/O requests via a socket. It issues individual disk I/Os directly to the attached SCSI disks using the Linux raw SCSI device `/dev/sg`. With an SMP host, the process can run on a separate CPU of the same host, to minimize the effect on the execution of the main application.

An application using *Atropos* is linked with a stub library providing API functions for reading and writing. The library uses shared memory to avoid data copies and communicates through the socket with the *Atropos* LVM process. The *Atropos* LVM organization is specified by a configuration file, which functions in lieu of a format command. The file lists the number of disks, *p*, the desired block size, *b*, and the list of disks to be used.

For convenience, the interface stub also includes three functions. The function *get_boundaries(LBN)* returns the stripe unit boundaries between which the given *LBN* falls. Hence, these boundaries form a collection of *w* contiguous *LBN*s for constructing efficient I/Os. The *get_rectangle(LBN)* function returns the *wp* contiguous *LBN*s in a single row across all disks. These functions are just convenient wrappers that calculate the proper *LBN*s from the *w* and *p* parameters. Finally, the stub interface also includes a *batch()* function to explicitly group READ and WRITE commands (e.g., for semi-sequential access).

With no outstanding requests in the queue (i.e., the disk is idle), current SCSI disks will immediately schedule the first received request of batch, even though it may not be the one with the smallest rotational latency. This diminishes the effectiveness of semi-sequential access. To overcome this problem, our *Atropos* implementation "pre-schedules" the batch of requests by sending first the request that will incur the smallest rotational latency. It uses known techniques for SPTF scheduling outside of disk firmware [14]. With the help of a detailed and validated model of the disk mechanics [2, 21], the disk head position is deduced from the location and time of the last-completed request. If disks waited for all requests of a batch before making a scheduling decision, this pre-scheduling would not be necessary.

Our implementation of the *Atropos* logical volume manager is about 2000 lines of C++ code and includes implementations of RAID levels 0 and 1. Another 600 lines of C code implement methods for automatically extracting track boundaries and head switch time [22, 26].

4 Efficient access in database systems

Efficient access to database tables in both dimensions can significantly improve performance of a variety of queries doing selective table scans. These queries can request (i) a subset of columns (restricting access along the primary dimension, if the order is column-major), which is prevalent in decision support workloads (TPC-H), (ii) a subset of rows (restricting access along the secondary dimension), which is prevalent in online transaction processing (TPC-C), or (iii) a combination of both.

A companion project [24] to *Atropos* extends the Shore database storage manager [3] to support a page layout that takes advantage of *Atropos*'s efficient accesses in both dimensions. The page layout is based on a cache-efficient page layout, called PAX [1], which extends the NSM page layout to group values of a single attribute into units called "minipages". Minipages in PAX exist to take advantage of CPU cache prefetchers to minimize cache misses during single-attribute memory accesses. We use minipages as well, but they are aligned and sized to fit into one or more 512 byte *LBN*s, depending on the relative sizes of the attributes within a single page.

The mapping of 8 KB pages onto the quadrangles of the *Atropos* logical volume is depicted in Figure 6. A single page contains 16 equally-sized attributes, labeled A1–A16, where each attribute is stored in a separate minipage that maps to a single *VLBN*. Accessing a single page is thus done by issuing 16 batched requests to every 16^{th} (or more generally, *wp*-th) *VLBN*. Internally, the *VLBN*s comprising this page are mapped diagonally to the blocks marked with the dashed arrow. Hence, 4 semi-sequential accesses proceeding in parallel can fetch the entire page (i.e., row-major order access).

Individual minipages are mapped across sequential runs of *VLBN*s. For example, to fetch attribute A1 for records 0–399, the database storage manager can issue one efficient sequential I/O to fetch the appropriate minipages. *Atropos* breaks this I/O into four efficient, track-based disk accesses proceeding in parallel. The database storage manager then reassembles these minipages into appropriate 8 KB pages [24].

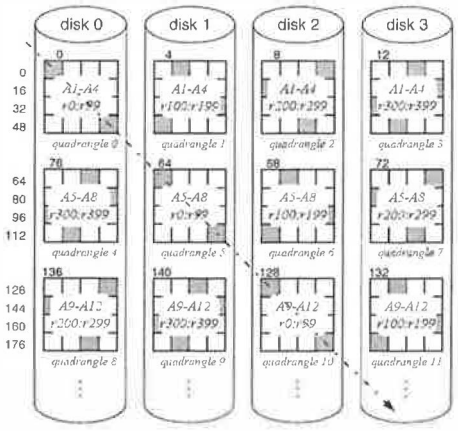


Figure 6: **Mapping of a database table with 16 attributes onto *Atropos* logical volume with 4 disks.**

Fetching any subset of attributes for a given page (record range) is thus a simple matter of issuing the corresponding number of I/Os, each accessing a contiguous region of the *VLBN* space mapped to a contiguous region on the disk. If several I/Os fall onto stripe units mapped to the same disk, the internal disk scheduler optimizes their execution by minimizing positioning times.

5 Evaluation

This section evaluates the performance of *Atropos*. First, it quantifies the efficiencies of sequential, semi-sequential and random accesses and shows the impact of disk trends on the layout parameter values. For all access patterns, *Atropos* achieves performance comparable or superior to conventional disk array data organizations. Second, trace replay experiments of a TPC-H workload on the *Atropos* implementation shows the benefit of matching the stripe-unit size to the exact track size and exposing it to applications. Third, the benefits of *Atropos*'s data organizations are shown for (a subset of) queries from the TPC-H benchmark running on the Shore database storage manager [3] and our *Atropos* implementation.

5.1 Experimental setup

The experiments were performed on a four-way 500 MHz Pentium III machine with 1 GB of memory running Linux kernel v. 2.4.7 of RedHat 7.1 distribution. The machine was equipped with two Adaptec Ultra160 Wide SCSI adapters on two separate PCI buses, each controlling two 36 GB Maxtor Atlas 10K III disks.

5.2 Quantifying access efficiency

Traditional striped layouts of data across disks in a RAID group offer efficient (sequential) access along one major. The efficiency of accessing data along the other major is much lower, essentially involving several random accesses. *Atropos*'s quadrangle layout, on the other hand, offers streaming efficiency for sequential accesses and much higher efficiency for the other-major access. We define "access efficiency" as the fraction of total access time spent reading/writing data from/to the media. The access efficiency is reduced by activities other than data transfer, including seeks, rotational latencies, and track switches. The efficiencies and response times described in this subsection are for a single disk. With p disks comprising a logical volume, each disk can experience the same efficiency while accessing data in parallel.

5.2.1 Efficient access in both majors

Figure 7 graphs the access efficiency of the quadrangle layout as a function of I/O size. It shows two important features of the *Atropos* design. First, accessing data in the primary-order (line 1) matches the best-possible efficiency of track-based access with traxtents. Second, the efficiency of the other other-major order access (line 2) is much higher than the same type of access with the *Naïve* layout of conventional disk arrays (line 3), thanks to semi-sequential access.

The data in the graph was obtained by measuring the response times of requests issued to randomly chosen *DLBN*s, aligned on track boundaries, within the Atlas 10K III's outer-most zone (686 sectors or 343 KB per track). The average seek time in the first zone is 2.46 ms. The drop in the primary-major access efficiency at the 343 KB mark is due to rotational latency and an additional track switch incurred for I/Os larger than the track size, when using a single disk.

The I/O size for the other-major access with the quadrangle layout is the product of quadrangle depth, d, and the number of consecutive *DLBN*s, b, accessed on each track. For $d = 4$, a request for S sectors is split into four I/Os of $S/4$ *DLBN*s. For this access in the *Naïve* layout (line 3), servicing these requests includes one seek and some rotational latency for each of the four b-sized I/Os, which are "randomly" located on each of the four consecutive tracks.

The efficiency of semi-sequential quadrangle access (line 2) with I/O sizes below 124 KB is only slightly smaller than that of the efficiency of track-based access with traxtents. Past this point, which corresponds to the one-revolution constraint, the efficiency increases at a slower rate, eventually surpassing the efficiency value at the 124 KB mark. However, this increase in efficiency comes at a cost of increased request latency; the access will now require multiple revolutions to service. The continuing increase in efficiency past the 124 KB mark is due to amortizing the cost of a seek by larger data transfer. Recall that each request includes an initial seek.

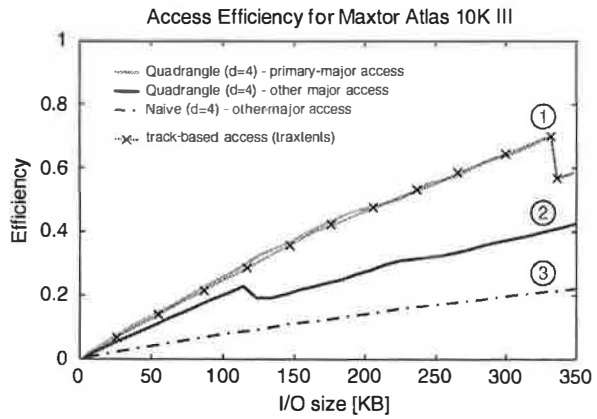


Figure 7: **Comparison of access efficiencies.** The maximal streaming efficiency, i.e., without seeks, for this disk is 0.82 (computed by Equation 6 in Appendix A).

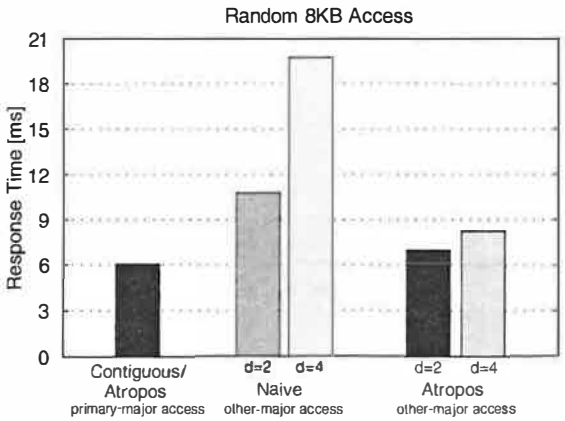


Figure 8: **Comparison of response times for random access.**

5.2.2 Random accesses in the other-major

Figure 8 compares access times for a random 8 KB chunk of data with different data layouts. The goal is to understand the cost of accessing data in the other-major order (e.g., row-major order access of the table in Figure 2). For context, a block in the primary-major has its data mapped to consecutive *LBN*s. Such an access incurs an average seek of 2.46 ms and an average rotational latency of half a revolution, followed by an 8 KB media access. The total response time of 5.93 ms is shown by the bar labeled "Contiguous."

Accessing 8 KB of data randomly spread across non-contiguous *VLBN*s (e.g., single row access in the *Naïve* layout of Figure 2) incurs nearly half of a revolution of rotational latency for each of the d accesses in addition to the same initial seek. Such an access results in a large response time, as shown by the bars labeled "Naive." Database systems using the DSM data layout decomposed into d separate tables suffer this high penalty when complete records are retrieved.

In contrast, with the quadrangle layout, an access in the other-major incurs only a single seek and much less total rotational latency than the access in the traditional *Naïve* layout. This access still incurs one (for $d = 2$) or three (for $d = 4$) track switches, which explains the penalty of 16% and 38% relative to the best case.

5.2.3 Access performance analysis

Using parameters derived in Section 3.2 and the analytical model described in Appendix A, we can express the expected response time for a quadrangle access and compare it with measurements taken from a real disk.

Figure 9 plots response times for quadrangle accesses to the disk's outer-most zone as a function of I/O request size, S, and compares the values obtained from the analytic model to measurements from a real disk. The close match between these data sets demonstrates that *Atropos* can reliably determine proper values of quadrangle layout analytically rather than empirically, which may be time consuming. The data is shown for the Atlas 10K III disk: $N = 686$, $H = 139$, and 6 ms revolution time.

The plotted response time does not include seek time; adding it to the response time would simply shift the lines up by an amount equivalent to the average seek time. The total I/O request size, S, shown along the x-axis is determined as $S = db$. With $d = 1$, quadrangle access reduces to normal disk access. Thus, the expected response time grows from 3 to 6 ms. For $d = 6$, the response time is at least 10.8 ms, even for the smallest possible I/O size, because $D_{max} = 5$ for the given disk.

The most prominent features of the graph are the steps from the 6 ms to 10–12 ms regions. This abrupt change in response time shows the importance of the one-revolution constraint. If this constraint is violated by an I/O size that is too large, the penalty in response time is significant.

The data measured on the real disk (dashed lines in Figure 9) match the predicted values. To directly compare the two sets of data, the average seek value was subtracted from the measured values. The small differences occur because the model does not account for bus transfer time, which does not proceed entirely in parallel with media transfer.

5.2.4 Effect of disk characteristics

Figure 9 shows the relationship between quadrangle dimensions and disk characteristics of one particular disk with $D_{max} = 5$. To determine how disk characteristics affect the quadrangle layout, we use the analytic model to study other disks. As shown in Table 2, the dimensions of the quadrangles mapped to the disks' outer-most zones remain stable across different disks of the past decade. The smaller D_{max} for the Atlas 10K III is due to an unfortunate track skew/head switch of $H = 139$. If $H = 136$, $D_{max} = 6$ and $b = 1$.

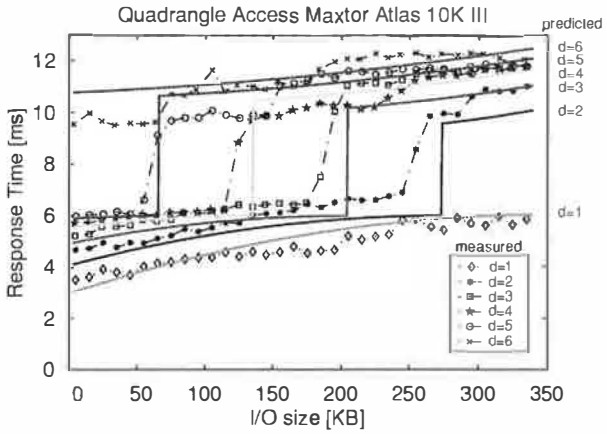


Figure 9: **Response time of semi-sequential access.**

Table 2 also shows that, with d set to D_{max}, the number of *DLBN*s, b, accessed at each disk track remains below 10, with the exception of the Atlas 10K III. The data reveals another favorable trend: the small value of R (number of *DLBN*s on each track not mapped to *VLBN*s) is a modest capacity tradeoff for large performance gains. With the exception of the Atlas 10K III disk, less than 1% of the total disk capacity would be wasted. For that disk, the value is 1.5%.

5.3 Track-sized stripe units

We now evaluate the benefits of one *Atropos* feature in isolation: achieving efficient sequential access by matching stripe units to exact track boundaries and exposing it to applications. To do so, we replayed block-level I/O traces of the TPC-H benchmark, representing a decision support system workload dominated by large sequential I/O. The original traces were captured on an IBM DB2 v. 7.2 system using 8 KB NSM pages and running each of the 22 TPC-H queries separately. The configuration specifics and the description of the trace replay transformations are detailed elsewhere [20].

For the experiments described in the remainder of this section, we used a single logical volume created from four disks ($p = 4$) and placed it on the disks' outermost zone, giving it a total size of 35 GB. The quadrangle layout was configured as RAID 0 with $d = 4$ and $b = 1$.

To simulate the effects of varying stripe unit size and exposing its value to DB2, we modified the captured traces by compressing back-to-back sequential accesses to the same table or index into one large I/O. We then split this large I/O into individual I/Os according to the stripe unit size, preserving page boundaries.

To simulate traditional disk arrays with a (relatively small) hard-coded stripe unit size, we set the stripe unit size to 64 KB (128 blocks) and called this base case scenario *64K-RAID*. To simulate systems that approximate track size, but do not take advantage of disk characteristics, we set the stripe unit to 256 KB (512 blocks) and called this scenario *Approximate-RAID*. By taking advantage of automatically-extracted explicit disk characteristics, *Atropos* can set the stripe unit size to the exact track size of 343 KB and we called this scenario *Atropos-RAID*. For all experiments, we used the *Atropos* LVM configured for RAID 0 (i.e., $d = 1$ and w was 128, 512, and 686 blocks respectively) and I/O sizes matching stripe unit sizes.

Disk	Year	D_{max}	b	R
HP C2247	1992	7	1	0
IBM Ultrastar 18 ES	1998	7	5	0
Quantum Atlas 10K	1999	6	2	0
Seagate Cheetah X15	2000	6	4	2
Seagate Cheetah 36ES	2001	6	7	3
Maxtor Atlas 10K III	2002	5	26	10
Seagate Cheetah 73LP	2002	7	8	2

Table 2: **Quadrangle parameters across disk generations.** For each disk, the amount of space not utilized due to R residual *DLBN*s is less than 1% with the exception of the Atlas 10K III, where it is 1.5%.

The resulting I/O times of the 22 TPC-H queries are shown in Figure 10. The graph shows the speedup of the *Approximate-RAID* and *Atropos-RAID* scenarios relative to the base case scenario *64K-RAID*. The results are in agreement with the expectations of sequential access efficiencies in Figure 7. The larger, more efficient I/Os of the *Approximate-RAID* and *Atropos-RAID* result in the observed speedup. The exact full-track access of *Atropos-RAID* provides additional 2%–23% benefit.

Some fraction of the query I/Os are not sequential or stripe-unit sized (e.g., less than 1% for query 1 vs. 93% and 99% for queries 14 and 17, respectively). These differences explain why some queries are sped up more than others; *Atropos* does not significantly improve the small random I/Os produced by index accesses.

The efficiency of *Atropos-RAID* is automatically maintained with technology changes (e.g., increasing numbers of sectors per track). While *Approximate-RAID* must be manually set to approximate track size (provided it is known in the first place), *Atropos* automatically determines the correct value for its disks and sets its stripe unit size accordingly, eliminating the error-prone manual configuration process.

5.4 Two-dimensional data access

To quantify the benefits of both efficient sequential and semi-sequential accesses, we used a TPC-H benchmark run on the Shore database storage manager [3] with three different layouts. The first layout, standard *NSM*, is optimized for row-major access. The second layout, standard *DSM*, vertically partitions data to optimize column-major access. The third layout, here called *AtroposDB*, uses the page layout described in Section 4, which can

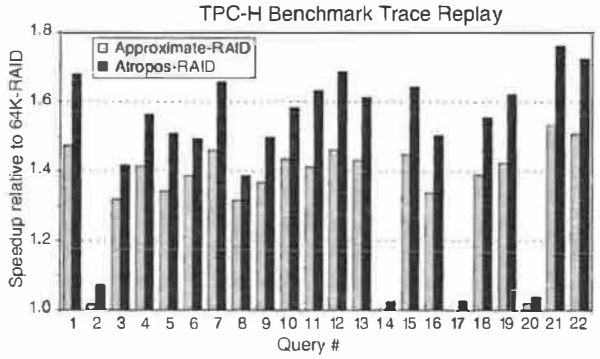


Figure 10: **TPC-H trace replay on *Atropos*.**

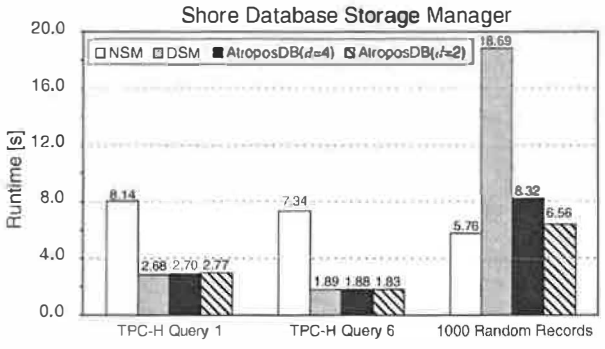


Figure 11: **Database access results.** This figure shows the runtime of TPC-H queries 1 and 6 for three different layouts. The *NSM* and *DSM* layouts are optimized for row- and column-order access respectively, trading off performance in the other-order. *Atropos*, on the other hand, offers efficient execution of TPC-H queries as well as random-page access in OLTP workloads (random access to 1000 records).

take advantage of *Atropos*'s full set of features. Each setup uses an 8 KB page size.

The results for TPC-H queries 1 and 6 are shown in Figure 11.[1] These queries scan through the LINEITEM table (the largest table of the TPC-H benchmark), which consists of 16 attributes, and calculate statistics based on a subset of six (Q1) and four (Q6) attributes. As expected, the performance of *DSM* and *AtroposDB* is comparable, since both storage models can efficiently scan through the table, requesting data only for the desired attributes. *NSM*, on the other hand, fetches pages that contain full records (including the attributes not needed by the queries), which results in the observed 2.5× to 4× worse performance. All scans were performed on a 1 GB TPC-H installation, with the LINEITEM table constituting about 700 MB.

Random record accesses to the LINEITEM table approximate an OLTP workload behavior, which is dominated by row-major access. For *DSM*, which is not suitable for row-major access, this access involves 16 random accesses to the four disks. For *NSM*, this access involves a random seek and rotational latency of half a revolution, followed by an 8 KB page access, resulting in a run time of 5.76 s. For *AtroposDB*, this access includes the same seek, but most rotational latency is eliminated, thanks to semi-sequential access. It is, however, offset by incurring additional track switches. With d=4, the run time is 8.32 s; with d=2, it is 6.56 s. These results are in accord with the random access results of Figure 8.

6 Related work

Atropos builds upon many ideas proposed in previous research. Our contribution is in integrating them into a novel disk array organization, cleanly extending the storage interface to allow applications to exploit it, and evaluating the result with a real implementation executing benchmark database queries.

[1]Of the four TPC-H queries implemented by the Shore release available to us, only Q1 and Q6 consist of LINEITEM table scan that is dominated by I/O time.

Atropos's track-based striping is inspired by recent work on track-aligned extents [22], which showed that track-based access could provide significant benefits for systems using a single disk. Most high-end systems use disk arrays and LVMs, and *Atropos* allows those to utilize track-based access.

Gorbatenko and Lilja [9] proposed diagonal disk layout of relational tables, allowing what we call semi-sequential access. *Atropos* integrates such diagonal layout with track-aligned extents to realize its support for two-dimensional data structures with no penalty to the primary access order.

Section 2.4 describes how modern databases address the trade-off between row-major and column-major access to database tables. Recently, Ramamurthy and DeWitt proposed that database systems should address this trade-off by maintaining two copies of data, one laid out in column-major order and the other in row-major order, to ensure efficient accesses in both dimensions [15]. This approach, however, not only doubles the storage requirements, but also makes updates difficult; they must propagate to two copies that are laid out differently. *Atropos* provides efficient accesses in both dimensions with only one copy of the data.

Denehy et al. [7] proposed the ExRAID storage interface that exposes some information about parallelism and failure-isolation boundaries of a disk array. This information was shown to allow application software to control data placement and to dynamically balance load across the independent devices in the disk array. ExRAID exposed coarse boundaries that were essentially entire volumes, which could be transparently spread across multiple devices. *Atropos*, on the other hand, defines a particular approach to spreading data among underlying devices and then exposes information about its specifics. Doing so allows applications to benefit from storage-managed parallelism and redundancy,

while optionally exploiting the exposed information to orchestrate its data layouts and access patterns.

7 Conclusions

The *Atropos* disk array LVM employs a new data organization that allows applications to take advantage of features built into modern disks. Striping data in track-sized units lets them take advantage of zero-latency access to achieve efficient access for sequential access patterns. Taking advantage of request scheduling and knowing exact head switch times enables semi-sequential access, which results in efficient access to diagonal sets of non-contiguous blocks.

By exploiting disk characteristics, a new data organization, and exposing high-level constructs about this organization, *Atropos* can deliver efficient accesses for database systems, resulting in up to 4× speed-ups for decision support workloads, without compromising performance of OLTP workloads.

Acknowledgements

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, LSI Logic, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. This work is funded in part by NSF grants CCR-0113660, IIS-0133686, and CCR-0205544, and by an IBM faculty partnership award.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. *International Conference on Very Large Databases* (Rome, Italy, 11–14 September 2001), pages 169–180. Morgan Kaufmann Publishing, Inc., 2001.

[2] J. S. Bucy and G. R. Ganger. *The DiskSim simulation environment version 3.0 reference manual*. Technical Report CMU–CS–03–102. Department of Computer Science Carnegie-Mellon University, Pittsburgh, PA, January 2003.

[3] M. J. Carey et al. Shoring up persistent applications. *ACM SIGMOD International Conference on Management of Data* (Minneapolis, MN, 24–27 May 1994). Published as *SIGMOD Record*, **23**(2):383–394, 1994.

[4] P. M. Chen and D. A. Patterson. Maximizing performance in a striped disk array. *ACM International Symposium on Computer Architecture* (Seattle, WA), pages 322–331, June 1990.

[5] G. P. Copeland and S. Khoshafian. A decomposition storage model. *ACM SIGMOD International Conference on Management of Data* (Austin, TX, 28–31 May 1985), pages 268–279. ACM Press, 1985.

[6] *IBM DB2 Universal Database Administration Guide: Implementation*. Document number SC09-2944-005.

[7] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. *Summer USENIX Technical Conference* (Monterey, CA, 10–15 June 2002), pages 177–190, 2002.

[8] G. R. Ganger. *Blurring the line between OSs and storage devices*. Technical report CMU–CS–01–166. Carnegie Mellon University, December 2001.

[9] G. G. Gorbatenko and D. J. Lilja. *Performance of two-dimensional data models for I/O limited non-numeric applications*. Laboratory for Advanced Research in Computing Technology and Compilers Technical report ARCTiC–02–04. University of Minnesota, February 2002.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2003.

[11] R. Y. Hou and Y. N. Patt. Track piggybacking: an improved rebuild algorithm for RAID5 disk arrays. *International Conference on Parallel Processing* (Urbana, Illinois), 14–18 August 1995.

[12] D. M. Jacobson and J. Wilkes. *Disk scheduling algorithms based on rotational position*. Technical report HPL–CSP–91–7. Hewlett-Packard Laboratories, Palo Alto, CA, 24 February 1991, revised 1 March 1991.

[13] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 69–77, 1987.

[14] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 275–288. USENIX Association, 2002.

[15] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. *International Conference on Very Large Databases* (Hong Kong, China, 20–23 August 2002), pages 430–441. Morgan Kaufmann Publishers, Inc., 2002.

[16] A. L. N. Reddy and P. Banerjee. A study of parallel disk organizations. *Computer Architecture News*, **17**(5):40–47, September 1989.

[17] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52. ACM Press, February 1992.

[18] K. Salem and H. Garcia-Molina. Disk striping. *International Conference on Data Engineering* (Los Angeles, CA), pages 336–342. IEEE, Catalog number 86CH2261-6, February 1986.

[19] J. Schindler. *Matching application access patterns to storage device characteristics*. PhD thesis. Carnegie Mellon University, 2004.

[20] J. Schindler, A. Ailamaki, and G. R. Ganger. Lachesis: robust database storage management based on device-specific performance characteristics. *International Conference on Very Large Databases* (Berlin, Germany, 9–12 September 2003). Morgan Kaufmann Publishing, Inc., 2003.

[21] J. Schindler and G. R. Ganger. *Automated disk drive characterization*. Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[22] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 259–274. USENIX Association, 2002.

[23] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. *Winter USENIX Technical Conference* (Washington, DC, 22–26 January 1990), pages 313–323, 1990.

[24] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. *Clotho: decoupling memory page layout from storage organization*. Technical report CMU–PDL–04–102. Carnegie-Mellon University, Pittsburgh, PA, February 2004.

[25] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.

[26] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada), pages 146–156, May 1995.

A Access Efficiency Calculations

Let $T(N,K)$ be the time it takes to service a request of K sectors that fit onto a single track of a disk with N sectors per track (i.e., track-aligned access). Ignoring seek, and assuming no zero-latency access, this time can be expressed as

$$T_{nzl}(N,K) = \frac{N-1}{2N} + \frac{K}{N}$$

where the first term is the average rotational latency, and the second term is the media access time. For disks with zero-latency access, the first term is not constant; rotational latency decreases with increasing K. Thus,

$$T_{zl}(N,K) = \frac{(N-K+1)(N+K)}{2N^2} + \frac{K-1}{N}$$

These expressions are derived elsewhere [19].

The efficiency of track-based access is the ratio between the raw one revolution time, T_{rev}, and the time it takes to read $S = kN$ sectors for some large k. Hence,

$$E_n = \frac{kT_{rev}}{T_n(N,N) + (k-1)(T_{hs} + T_{rev})} \approx \frac{kT_{rev}}{k(T_{hs} + T_{rev})}$$

where $T_n(N,N)$ is the time to read data on the first track, and $(k-1)(T_{hs}+T_{rev})$ is the time spent in head switches and accessing the remaining tracks. In the limit, the access efficiency is

$$E_n(N,H) = 1 - \frac{H}{N} \tag{6}$$

which is the maximal streaming efficiency of a disk.

The maximal efficiency of semi-sequential quadrangle access is simply

$$E_q(N,H) = \frac{T_{rev}}{T_q(N,S)} = \frac{T_{rev}}{T_{zl}(N, db + (d-1)H)} \tag{7}$$

with d and b set accordingly.

B Relaxing the one-revolution constraint

Suppose that semi-sequential access to d blocks, each of size b, from a single quadrangle takes more than one revolution. Then the inequality in Equation 1 will be larger than 1. With probability $1/N$, a seek will finish with disk heads positioned exactly at the beginning of the b sectors mapped to the first track (the upper left corner of the quadrangle in Figure 12). In this case, the disk will access all db sectors with maximal efficiency (only incurring head switch of H sectors for every b-sector read).

However, with probability $1 - 1/N$, the disk heads will land somewhere "in the middle" of the b sectors after a seek, as illustrated by the arrow in Figure 12. Then, the access will incur a small rotational latency to access the beginning of the nearest b sectors, which are, say, on the k-th track. After this initial rotational latency, which is, on average, equal to $(b-1)/2N$, the $(d-k)b$ sectors mapped onto $(d-k)$ tracks can be read with maximal efficiency of the semi-sequential quadrangle access.

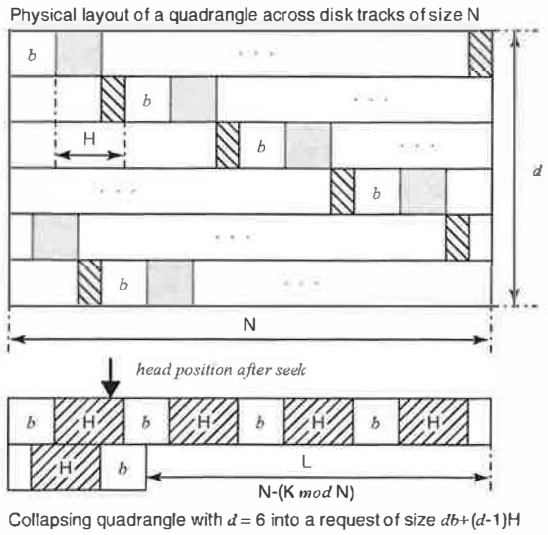


Figure 12: **An alternative representation of quadrangle access.**

To read the remaining k tracks, the disk heads will need at be positioned to the beginning of the b sectors on the first track. This will incur a small seek and additional rotational latency of L/N. Hence, the resulting efficiency is much lower than when the one-revolution constraint holds, which avoids this rotational latency.

We can express the total response time for quadrangle access without the one-revolution constraint as

$$T_q(N,S) = \frac{b-1}{2N} + \frac{K}{N} + P_{lat}\frac{L}{N} \tag{8}$$

where $P_{lat} = (N-H-b-1)/N$ is the probability of incurring the additional rotational latency after reading k out of d tracks, $K = db - (d-1)H$ is the effective request size, $L = N - (K \bmod N)$, and $S = db$ is the original request size. To understand this equation, it may be helpful to refer to the bottom portion of Figure 12.

The efficiencies of the quadrangle accesses with and without the one-revolution constraint are approximately the same when the time spent in rotational latency and seek for the unconstrained access equals to the time spent in rotational latency incurred during passing over dR residual sectors. Hence,

$$\frac{dR}{N} = \frac{N-1}{N}\left(\frac{N-1}{2N} + Seek\right)$$

Ignoring seek and approximating $N-1$ to be N, this occurs when $R \neq 0$ and

$$d \approx \frac{N}{2R}.$$

Thus, in order to achieve the same efficiency for the non-constrained access, we will have to access at least d *VLBN*s. However, this will significantly increase I/O latency. If $R = 0$ i.e., when there are no residual sectors, the one-revolution constraint already yields the most efficient quadrangle access.

C-Miner: Mining Block Correlations in Storage Systems

Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan and Yuanyuan Zhou

Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801
{zli4, zchen9, smsriniv, yyzhou}@cs.uiuc.edu

ABSTRACT

Block correlations are common semantic patterns in storage systems. These correlations can be exploited for improving the effectiveness of storage caching, prefetching, data layout and disk scheduling. Unfortunately, information about block correlations is not available at the storage system level. Previous approaches for discovering file correlations in file systems do not scale well enough to be used for discovering block correlations in storage systems.

In this paper, we propose *C-Miner*, an algorithm which uses a data mining technique called *frequent sequence mining* to discover block correlations in storage systems. *C-Miner* runs reasonably fast with feasible space requirement, indicating that it is a practical tool for dynamically inferring correlations in a storage system. Moreover, we have also evaluated the benefits of block correlation-directed prefetching and data layout through experiments. Our results using real system workloads show that correlation-directed prefetching and data layout can reduce average I/O response time by 12-25% compared to the base case, and 7-20% compared to the commonly used sequential prefetching scheme.

1 Introduction

To satisfy the growing demand for storage, modern storage systems are becoming increasingly intelligent. For example, the IBM Storage Tank system [29] consists of a cluster of storage nodes connected using a storage area network. Each storage node includes processors, memory and disk arrays. An EMC Symmetric server contains up to *eighty* 333 MHz microprocessors with up to 4-64 GB of memory as the storage cache [19]. Figure 1 gives an example architecture of modern storage systems. Many storage systems also provide virtualization capabilities to hide disk layout and configurations from storage clients [36, 3].

Unfortunately, it is not an easy task to exploit the increasing intelligence in storage systems. One primary reason is the narrow I/O interface between storage applications and storage systems. In such a simple interface, storage applications perform only block read or write operations without any indication of access patterns or data semantics. As a result, storage systems can only manage data at the block level without knowing any semantic information such as the semantic correlations between blocks. Therefore, much previous work had to rely on simple patterns such as temporal locality, sequentiality, and loop references to improve storage system performance, without fully exploiting its intelligence. This motivates a more powerful analysis tool to discover more complex patterns, especially semantic patterns, in storage systems.

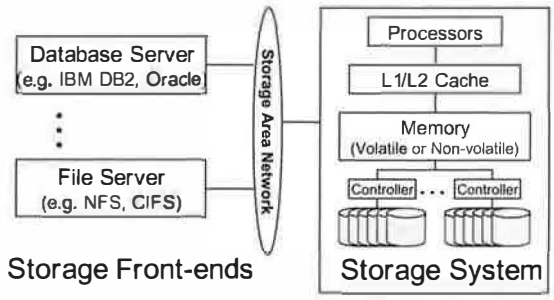


Figure 1: Example of modern storage architecture

Block correlations are common semantic patterns in storage systems. Many blocks are correlated by semantics. For example, in a database that uses index trees such as B-trees to speed up query performance, a tree node is correlated to its parent node and its ancestor nodes. Similarly, in a file server-backend storage system, a file block is correlated to its inode block. Correlated blocks tend to be accessed relatively close to each other in an access stream. Exploring these correlations is very useful for improving the effectiveness of storage caching, prefetching, data layout and disk scheduling. For example, at each access, a storage system can prefetch correlated blocks into its storage cache so that subsequent accesses to these blocks do not need to access disks, which is several orders of magnitude slower than accessing directly from a storage cache. As self-managing systems are becoming ever so important, capturing block correlations would enhance the storage system's knowledge about its workloads, a necessary step toward providing self-tuning capability.

Unfortunately, information about block correlations are unavailable at a storage system because a storage system exports only block interfaces. Since databases or file systems are typically provided by vendors different from those of storage systems, it is quite difficult and complex to extend the block I/O interface to allow upper levels to inform a storage system about block correlations. Recently, Arpaci-Dusseau et al. proposed a very interesting approach called semantically-smart disk systems (SDS) [54] by using a "gray-box" technology to infer data structure and categorize data in storage systems. However, this approach requires probing in the front-end and assumes that

the front-ends conform to the FFS-like file system layout.

An alternative approach is to infer block correlations fully transparently inside a storage system by only observing access sequences. This approach does not require any probing from a front-end and also makes no assumption about the type of the front-ends. Therefore, this approach is more general and can be applied to storage systems with any front-end file systems or database servers. Semantic distances [34, 35] and probability graphs [24, 25] are such "black-box" approaches. They are quite useful in discovering file correlations in file systems (see section 2.3 for more details).

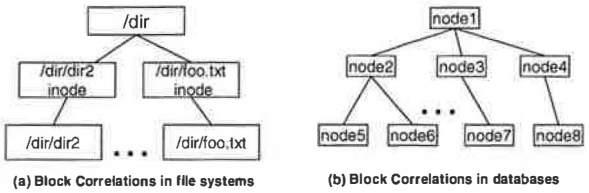
This paper proposes *C-Miner*, a method which applies a data mining technique called *frequent sequence mining* to discover block correlations in storage systems. Specifically, we have modified a recently proposed data mining algorithm called CloSpan [66] to find block correlations in several storage traces collected in real systems. To the best of our knowledge, *C-Miner* is the first approach to infer block correlations involving multiple blocks. Furthermore, *C-Miner* is more scalable and space-efficient than previous approaches. It runs reasonably fast with reasonable space overhead, indicating that it is a practical tool for dynamically inferring correlations in a storage system. Moreover, we have also evaluated the benefits of block correlation-directed prefetching and disk data layout using the real system workloads. Compared to the base case, this scheme reduces the average response time by 12% to 25%. Compared to the sequential prefetching scheme, it also reduces the average response time by 7% to 20%.

The paper is organized as follows. In the next section, we briefly describe block correlations, the benefits of exploiting block correlations, and approaches to discover block correlations. In section 3, we present our data mining method to discover block correlations. Section 4 discusses how to take advantage of block correlations in the storage cache for prefetching and disk layout. Section 5 presents our experimental results. Section 6 discusses the related work and section 7 concludes the paper.

2 Block Correlations

2.1 What are Block Correlations?

Block correlations commonly exist in storage systems. Two or more blocks are correlated if they are "linked" together semantically. For example, Figure 2(a) shows some block correlations in a storage system which manages data for an NFS server. In this example, a directory block "/dir" is directly correlated to the inode block of "/dir/foo.txt", which is also directly correlated to the file block of "/dir/foo.txt". Besides direct correlations, blocks can also be correlated indirectly through another block. For example, the directory block "/dir" is indirectly correlated to the file block of "/dir/foo.txt". Figure 2(b) shows block correlations in a database-backend storage system. Databases commonly use a tree structure such as B-tree or B*-tree to store data. In such a data structure, a node is directly correlated to its parent and children, and also indirectly correlated to its ancestor and descendant nodes.



(a) Block Correlations in file systems (b) Block Correlations in databases

Figure 2: Examples of block correlations

Unlike other access patterns such as temporal locality, block correlations are inherent in the data managed by a storage system. Access patterns such as temporal locality or sequentiality depend on workloads and can therefore change dynamically, whereas block correlations are relatively more stable and do not depend on workloads, but rather on data semantics. When block semantics are changed (for example, a block is reallocated to store other data), some block correlations may be affected. In general, block semantics are more stable than workloads, especially in systems that do not have very bursty deletion and insertion operations that can significantly change block semantics. As we will show in section 5.3, block correlations can remain stable for several days in file systems.

Correlated blocks are usually accessed very close to each other. This is because most storage front-ends (database servers or file servers) usually follow semantic "links" to access blocks. For example, an NFS server needs to access an inode block before accessing a file block. Similarly, a database server first needs to access a parent node before accessing its children. Due to the interleaving of requests and transactions, these I/O requests may not be always consecutive in the access stream received by a storage system. But they should be relatively close within a short distance from each other.

Spatial locality is a simple case of block correlations. An access stream exhibits spatial locality if, after a block is accessed, other blocks that are near it are likely to be accessed in the near future. This is based on the observation that a block is usually semantically correlated to its neighboring blocks. For example, if a file's blocks are allocated in disks consecutively, these blocks are correlated to each other. Therefore, in some workloads, these blocks are likely accessed one after another.

However, many correlations are more complex than spatial locality. For example, in an NFS server, an inode block is usually allocated separately from its file blocks and a directory block is allocated separately from the inode blocks of the files in this directory. Therefore, although accesses to these correlated blocks are close to each other in the access stream, they do not exhibit good spatial locality because these blocks are far away from each other in the disk layout and even on different disks.

In some cases, a block correlation may involve more than two blocks. For example, a three-block correlation might be: if *both* a and b are accessed recently, c is very likely to be accessed in a short period of time. Basically, a and b are correlated to c, but a or b alone may not be correlated to c. To give a real instance of this multi-block correlation, let us consider a B* tree which also links all the leaf nodes together. a, b and c

are all leaf nodes. If a is accessed, the system cannot predict that c is going to be accessed soon. However, if a *and* b are accessed one after another, it is likely that c will be accessed soon because it is likely that the front-end is doing a sequence scan of all the leaf nodes, which is very common in decision-support system (DSS) workloads [7, 68].

2.2 Exploiting Block Correlations

Block correlations can be exploited to improve storage system performance. First, correlations can be used to direct prefetching. For example, if a strong correlation exists between blocks a and b, these two blocks can be fetched together from disks whenever one of them is accessed. The disk read-ahead optimization is an example of exploiting the simple sequential block correlations by prefetching subsequent disk blocks ahead of time. Several studies [55, 14, 31] have shown that using even these simple sequential correlations can significantly improve the storage system performance. Our results in section 5.5 demonstrate that prefetching based on block correlations can improve the performance much better than such simple sequential prefetching in most cases.

A storage system can also lay out data in disks according to block correlations. For example, a block can be collocated with its correlated blocks so that they can be fetched together using just one disk access. This optimization can reduce the number of disk seeks and rotations, which dominate the average disk access latency. With correlation-directed disk layouts, the system only needs to pay a one-time seek and rotational delay to get multiple blocks that are likely to be accessed soon. Previous studies [52, 54] have shown promising results in allocating correlated file blocks on the same track to avoid track-switching costs.

Correlations can also be used to direct storage caching. For example, a storage cache can "promote" or "demote" a block after its correlated block is accessed or evicted. After an access to block A, blocks that are correlated to A are likely to be accessed very soon. Therefore, a cache replacement algorithm can specially "mark" these blocks to avoid being evicted. Similarly, after a block A is evicted, blocks that are correlated to A are not very likely to be accessed soon so it might be OK to also evict these blocks in subsequent replacement decisions. The storage cache can also give higher priority to those blocks that are correlated to many other blocks. Therefore, for databases that use tree structures, it would achieve a similar effect as the DBMIN cache replacement algorithm that is specially designed for database workloads [15]. This algorithm gives higher priority to root blocks or high-level index blocks to stay in a database buffer cache.

Besides performance, block correlations can also be used to improve storage system security, reliability and energy-efficiency. For example, malicious clients accesses the storage system in a very different pattern from the normal clients. By catching abnormal block correlations in an access stream, the storage system can detect such kind of malicious users. When a file block is archived to a tertiary storage, its correlated blocks may also need to be backed up in order to provide consistency. In addition, storage power management schemes can also take advantage of block correlations by clustering correlated blocks in the same disk so it is possible for other disks to transition into standby mode [11].

The experiments in this study focus on demonstrating the benefits of exploiting block correlations in improving storage system performance. The usages for security, reliability and energy-efficiency remain as our future work.

2.3 Obtaining Block Correlations

There can be three possible approaches to obtain block correlations in storage systems. These approaches trade transparency and generality for accuracy at different degrees. The "black box" approach is most transparent and general because it infers block correlations without any assumption or modification to storage front-ends. The "gray box" approach does not need modifications to front-end software but makes certain assumptions about front-ends and also requires probing from front-ends. The "white box" approach completely relies on front-ends to provide information and therefore has the most accurate information but is least transparent.

"Black Box" approaches infer block correlations completely inside a storage system, without any assumption on the storage front-ends. One commonly used method of this approach is to infer block correlations based on accesses. The observation is that correlated blocks are usually accessed relatively close to each other. Therefore, if two blocks are almost always accessed together within a short access distance, it is very likely that these two blocks are correlated to each other. In other words, it is possible to automatically infer block correlations in a storage system by dynamically analyzing the access stream.

In the field of networked or mobile file systems, researchers have proposed semantic distance (SD) [34, 35] or probability graphs [24, 25] to capture file correlations in file systems. The main idea is to use a graph to record the number of times two items are accessed within a specified access distance. In an SD graph, a node represents an accessed item B_1 with edges linking to other items. The weight of each edge (B_1, B_2) is the number of times that B_2 is accessed within the specified lookahead window of B_1's access. So if the weight for an edge is large, the corresponding items are probably correlated.

The algorithm to build the SD graph from an access stream works like this: Suppose the specified lookahead window size is 100, i.e., accesses that are less than 100 accesses apart are considered to be "close" accesses. Initially the probability graph is empty. The algorithm processes each access one after another. The algorithm always keeps track of the items of most recent 100 accesses in the current sliding window. When an item B is accessed, it adds node B into the graph if it is not in the graph yet. It also increments the weight of the edge (B_i, B) for any B_i accessed during the current window. If such an edge is not in the graph, it adds this edge and sets the initial weight to be 1. After the entire access stream is processed, the

algorithm rescans the SD graph and only records those correlations with weights larger than a given threshold.

Even though probability graphs or SD graphs work well for inferring file correlations in a file system, they, unfortunately, are not practical for inferring block correlations in storage systems because of two reasons. (1) *Scalability problem*: a semantic distance graph requires one node to represent each accessed item and also one edge to capture each non-zero-weight correlation. When the system has a huge number of items as in a storage system, an SD graph is too big to be practical. For instance, if we assume the specified window size is 100, it may require more than 100 edges associated with each node. Therefore, one node would occupy at least $100 \times 8 = 800$ (assuming each edge requires 8 bytes to store the weight and the disk block number of B_2). For a small storage system with only 80 GB and a block size of 8 KB, the probability graph would occupy 8 GB, 10% of the storage capacity. Besides space overheads, building and searching such a large graph would also take a significantly large amount of time. (2) *Multi-block correlation problem*: these graphs cannot represent correlations that involve more than two blocks. For example, the block correlations described at the end of the Section 2.1 cannot be conveniently represented in a semantic distance graph. Therefore, these techniques can lose some important block correlations.

In this paper, we present a practical black box approach that uses a data mining method to automatically infer both dual-block and multi-block correlations in storage systems. In Section 3, we describe our approach in detail.

"Gray Box" approaches are investigated by Arpaci-Dusseau et al in [5]. They developed three gray-box information and control layers between a client and the OS, and combined algorithmic knowledge, observations and inferences to collect information.

The gray-box idea has been explored by Sivathanu et al in storage systems to automatically obtain file-system knowledge [54]. The main idea is to probe from a storage front-end by performing some standard operations and then observing the triggered I/O accesses to the storage system. It works very well for file systems that conform to FFS-like structure (if the front-end security is not a concern). The advantage of this approach is that it does not require any modification to the storage front-end software. The tradeoff is that it requires the front-end to conform to specific disk layouts such as FFS-like structure.

"White Box" approaches rely on storage front-ends to directly pass semantic information to obtain block correlations in a storage system. For example, the storage I/O interface can be modified using a higher-level, object-like interface [23] so that correlations can be easily expressed using the object interface. The advantage with this approach is that it can obtain very accurate information about block correlations from storage front-ends. However, it requires modifying storage front-end software, some of which, such as database servers, are too large to be easily ported to object-based storage interface.

3 Mining for Block Correlations

Data mining, also known as knowledge discovery in databases (KDD), has developed quickly in recent years due to the wide availability of voluminous data and the imminent need for extracting useful information and knowledge from them. Traditional methods of data analysis dependent on human handling cannot scale well to huge sizes of data sets. In this section, we first introduce some fundamental data mining concepts and analysis methods used in our paper and then describe *C-Miner*, our algorithm for inferring block correlations in storage systems.

3.1 Frequent Sequence Mining

Different patterns can be discovered by different data mining techniques, including association analysis, classification and prediction, cluster analysis, outlier analysis, and evolution analysis [27]. Among these techniques, association analysis can help discover correlations between two or more sets of events or attributes. Suppose there exists a strong association between events x and y, it means that if event x happens, event y is also very likely to happen. We use the association rule $x \rightarrow y$ to describe such a correlation between these two events.

Frequent sequence mining is one type of association analysis to discover frequent subsequences in a sequence database [1]. A subsequence is considered *frequent* when it occurs in at least a specified number of sequences (called *min support*) in the sequence database. A subsequence is not necessarily contiguous in an original sequence. For example, a sequence database D has five sequences:

$$D = \{abced, abcef, agbch, abijc, aklc\}$$

The number of occurrences of subsequence abc is 4. We denote the number of occurrences of a subsequence as its *support*. Obviously, the smaller $min\ sup$ is, the more frequent subsequences the database contains. In the above example, if $min\ sup$ is specified as 5, only the subsequence ac is frequent; if $min\ sup$ is specified as 4, the frequent subsequences are $\{ab$: 4, ac: 5, bc: 4, abc: $4\}$, where the numbers are the supports of the subsequences.

Frequent sequence mining is an active research topic in data mining [67, 46, 6] with broad applications, such as mining motifs in DNA sequences, analysis of customer shopping sequences etc. To the best of our knowledge, our study is the first one that uses frequent sequence mining to discover patterns in storage systems.

C-Miner is based on a recently proposed frequent sequence mining algorithm called *CloSpan* (Closed Sequential Pattern mining)[66]. The main idea of CloSpan is to find only closed frequent subsequences. A *closed sequence* is a subsequence whose support is different from that of its super-sequences. In the above example, subsequence ac is closed because its support is 5, and the support of any one of its super-sequences (for example, abc and agc, etc.) is no more than 4; on the other

hand, subsequence ab is not closed because its support is the same as that of one of its super-sequences, abc.

CloSpan only produces the closed frequent subsequences rather than all frequent subsequences since any non-closed subsequences can be indicated by their super-sequences with the same support. In the above example, the frequent subsequences are $\{a$: 4, b: 4, c: 5, ab: 4, ac: 5, bc: 4, abc: $4\}$, but we only need to produce the closed subsequences $\{ac$: $5, abc$: $4\}$. This feature significantly reduces the number of patterns generated, especially for long frequent subsequences. More details can be found in [46, 66].

3.2 *C-Miner*: Our Mining Algorithm

Frequent sequence mining is a good candidate for inferring block correlations in storage systems. One can map a block to an item, and an access sequence to a sequence in the sequence database. Using frequent sequence mining, we can obtain all the frequent subsequences in an access stream. A frequent subsequence indicates that the involved blocks are frequently accessed together. In other words, frequent subsequences are good indications of block correlations in a storage system.

One limitation with the basic mining algorithm is that it does not consider the gap of a frequent subsequence. If a frequent sequence contains two accesses that are very far from each other in terms of access time, such a correlation is not interesting for our application. From the system's point of view, it is much more interesting to consider frequent access subsequences that are not far apart. For example, if a frequent subsequence xy always appears in the original sequence with a distance of more than 1000 accesses, it is not a very interesting pattern because it is hard for storage systems to exploit it. Further, such correlations are generally less accurate.

To address this issue, *C-Miner* restricts access distances. In order to describe how far apart two accesses are in the access stream, the access distance between them is denoted as gap, measured by the number of accesses between these two accesses. We specify a maximum distance threshold, denoted as $max\ gap$. All the uninteresting frequent sequences whose gaps are larger than the threshold are filtered out. This is similar to the lookahead window used in the semantic distance algorithms.

3.2.1 *Preprocessing*

Existing frequent sequence mining algorithms including CloSpan are designed to discover patterns for a sequence database rather than a single long sequence of time-series information as in storage systems. To overcome this limitation, *C-Miner* preprocesses the access sequence (that is, the history access trace) by breaking it into fixed-size short sequences. The size of each short sequence is called *cutting window size*.

There are two ways to cut the long access stream into short sequences - overlapped cutting and non-overlapped cutting. The overlapped cutting divides an entire access stream into many short sequences and leaves some overlapped regions between any two consecutive sequences. Non-overlapped cutting is straightforward; it simply splits the access stream into access sequences of equal size.

Figure 3 illustrates how these two methods cut the access stream $abcabdabeabf$ into short sequences with length of 4. Overlapped cutting may increase the number of occurrences for some subsequences if it falls in the overlapped region. In the example shown in Figure 3, using overlapped cutting results in 5 short sequences: $\{abca, cabd, bdab, abea, eabf\}$. The subsequences ab in $bdab$ and $abea$ occurs only once in the original access stream, but now is counted twice since the short sequences, $bdab$ and $abea$, overlap with each other. It is quite difficult to determine how many redundant occurrences there are due to overlapping. Another drawback is that the overlapped cutting generates more sequences than non-overlapped cutting. Therefore it takes the mining algorithm a longer time to infer frequent subsequences.

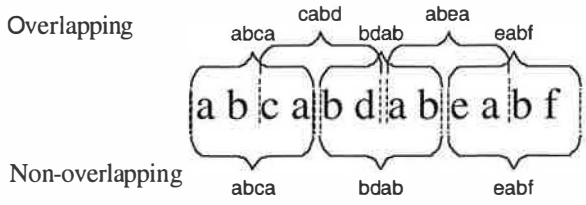


Figure 3: Overlapping and non-overlapping window (Cutting window size is 4)

Using non-overlapped cutting can, however, lead to loss of frequent subsequences that are split into two or more sequences, and therefore can decrease the support values of some frequent subsequences because some of their occurrences are split into two sequences. In the example shown in figure 3, the non-overlapped cutting results in only 3 sequences: $\{abca, bdab, eabf\}$. The support for ab is 3, but the actual support in the original long sequence is 4. The lost support is because the second occurrence is broken across two windows and is therefore not counted.

But we believe that the amount of lost information in the non-overlapped cutting scheme is quite small, especially if the cutting window size is relatively large. Since *C-Miner* restricts the access distance of a frequent subsequence, only a few frequent subsequences may be split across multiple windows. Suppose the instances of a frequent subsequence are distributed uniformly in the access stream, the cutting windows size is w and the maximum access distance for frequent sequences is $max\ gap$ ($max\ gap \ll w$). Then, in the worst case, the probability that an instance of a frequent subsequence is split across two sequences is $max\ gap/w$. For example, if the access distance is limited within 50, and the cutting window size is 500 accesses, the support value is lost by at most 10% in the worst case. Therefore, most frequent subsequences would still be considered frequent after non-overlapped cutting. Based on this analysis, we use non-overlapped cutting in our experiments.

3.2.2 Core Algorithm

Once it has a database of short sequences, *C-Miner* mines the database and produces frequent subsequences, which can then be used to derive block correlations. *C-Miner* mainly consists of two stages: (1) generating a candidate set of frequent subsequences that includes all the closed frequent subsequences; and (2) pruning the non-closed subsequences from the candidate set.

In the first stage, *C-Miner* generates a candidate set of frequent sequences using a depth-first search procedure. The following pseudo-code shows the mining algorithm. In the algorithm, D_s is a suffix database which contains all the maximum suffixes of the sequences that contain the frequent subsequence s. For example, in the previous sequence database D, the suffix database of frequent subsequences ab is $D_{ab} = \{ced, cef, ch, ijc\}$.

Algorithm: MINING(s, D_s, min_sup, L)
Input: A frequent subsequence s,
a set of subsequence D_s,
support threshold min_sup.
Output: The frequent sequence set L.
1: insert s to L.
2: scan D_s to find every frequent item α such that $s \diamond \alpha$ is frequent sequence, $D_{s\diamond\alpha} \leftarrow$ { all maximum suffixes that can be concatenated with $s \diamond \alpha$}.
3: for each α do
MINING($s \diamond \alpha, D_{s\diamond\alpha}, min\_sup, L$).

(Note: $s \diamond \alpha$ means to concatenate s with α.)

There are two main ideas in *C-Miner* to improve the mining efficiency. The first idea is based on an obvious observation that if a sequence is frequent, then all of its subsequences are frequent. For example, if a sequence abc is frequent, all of its subsequences $\{a, b, c, ab, ac, bc\}$ are frequent. Based on this observation, *C-Miner* recursively produces a longer frequent subsequence by concatenating every frequent item to a shorter frequent subsequence that has already been obtained in the previous iterations.

To better explain this idea, let us consider an example. In order to get the set L_n of frequent subsequences with length n, we can join the set L_{n-1} of frequent subsequences with length $n-1$ and the set L_1 of frequent subsequences with length 1. For example, suppose we have already computed L_1 and L_2 as shown below. In order to compute L_3, we can first compute L'_3 by concatenating a subsequence from L_2 and an item from L_1:

$$\begin{aligned} L_1 &= \{a, b, c\}; \\ L_2 &= \{ab, ac, bc\}; \\ L'_3 &= L_2 \times L_1 \\ &= \{abc, abb, abc, aca, acb, acc, bca, bcb, bcc\} \end{aligned}$$

For greater efficiency, *C-Miner* does not join the sequences in set L_2 with all the items in L_1. Instead, each sequence in L_2 is concatenated with only the frequent items in its suffix database. In our example, for the frequent sequence ab in L_2, its suffix database is $D_{ab} = \{ced, cef, ch, ijc\}$, and only c is the frequent item, so ab is only concatenated with c and then we get a longer sequence abc that belongs to L'_3.

The second idea is used for efficiently evaluating whether a concatenated subsequence is frequent or not. It tries to avoid searching through the whole database. Instead, it only checks with certain suffixes. In the above example, for each sequence s in L'_3, *C-Miner* checks whether it is frequent or not by searching the suffix database D_s. If the number of its occurrences is greater than min_sup, s is added into L_3, which is the set of frequent subsequences of length 3. *C-Miner* continues computing L_4 from L_3, L_5 from L_4, and so on until no more subsequences can be added into the set of frequent subsequences.

In order to mine frequent sequences more efficiently, *C-Miner* uses a technique that can efficiently determine whether there are new closed patterns in search subspaces and stop checking those unpromising subspaces. The basic idea is based on the following observation about a closed sequence property. In the algorithm step 2, among all the sequences in D_s, if an item a always occurs before another item b, *C-Miner* does not need to search any sequences with prefix $s \diamond b$. The reason is that $\forall\gamma$, $s \diamond b \diamond \gamma$ is not closed under this condition. Take the previous sequence database as an example. a always occurs before b, so any subsequence with prefix b is not closed because it is also a subsequence with prefix ab. Therefore, *C-Miner* does not need to search the frequent sequences with prefix b because all these frequent sequences are included in the frequent sequences with prefix ab (e.g., bc is included in abc with support 4). Without searching these unpromising branches, *C-Miner* can generate the candidate frequent sequences much more efficiently.

3.2.3 Generating Association Rules

C-Miner produces frequent sequences that indicate block correlations, but it does not directly generate the association rules in the form of $x_1 x_2 \dashrightarrow y$, which is much easier to use in storage systems.

In order to convert the frequent sequences into association rules, *C-Miner* breaks each sequence into several rules. In order to limit the number of rules, *C-Miner* constrains the length of a rule (the number of items on the left side of a rule). For example, a frequent sequence abc may be broken into the following set of rules with the same support of abc:

$$\{a \dashrightarrow b, a \dashrightarrow c, b \rightarrow c, ab \dashrightarrow c\}$$

Different closed frequent sequences can be broken into the same rules. For example, both abc and abd can be broken into the same rule $a \rightarrow b$, but they may have different support values. The support of a rule is the *maximum* support of all corresponding closed frequent sequences.

3.2.4 Confidence of Rules

For each association rule, we also need to evaluate its accuracy . In order to describe the reliability of a rule, we introduce $confidence$ to measure the accuracy. For example, in the above example, a occurs 5 times, but ab only occurs 4 times; this means that when a is accessed, b is also accessed in the near future (within max_gap distance) with probability 80%. We call this probability the confidence of the rule. When we use an association rule to predict future accesses, its confidence indicates the expected prediction accuracy. Predictions based on low-confidence rules are likely to be wrong and may not be able to improve system performance. Worse still, they may hurt the system performance due to overheads and side-effects. Because of this, we use confidence to restrict rules and filter out those with low probability.

The $support$ metric is different from $confidence$. For example, suppose x and y are accessed only once in the entire access stream and their accesses are within the $max\ gap$ distance, the confidence of the association rule $x \rightarrow y$ is 100% whereas its support is only 1. This rule is not very interesting because it happens rarely. On the other hand, if a rule has high support but very low confidence (e.g. 5%), it may not be useful because it is too inaccurate to be used for prediction. Therefore, in practice, we usually specify a minimum support threshold $min\ sup$ and a minimum confidence threshold min_conf in order to filter low-quality association rules.

We can estimate the confidence for each rule in a simple way. Suppose we need to compute the confidence for rule $a \rightarrow b$. Assume that the supports for a and b are $sup(a)$ and $sup(b)$, respectively. Then the confidence for this rule is $sup(b)/sup(a)$. Since both sides of each rule are frequent sequences (or frequent items) and the supports for all the frequent sequences are already obtained from post-processing, $sup(a)$ and $sup(b)$ are ready to be used for computing the confidence of the rule.

3.3 Efficiency of *C-Miner*

Compared with other methods such as probability graphs or SD graphs, *C-Miner* can find more correlations, especially those multi-block correlations. From our experiments, we find that these multi-block correlations are very useful for systems.

For dual-block correlations, which can also be inferred using previous approaches, *C-Miner* is more efficient. First, *C-Miner* is much more space efficient than SD graphs because it does not need to maintain the information for non-frequent sequences, whereas SD graphs need to keep the information for every block during the graph building process. Second, in terms of time complexity, *C-Miner* is the same ($O(n)$) as SD. But in practice, since *C-Miner* has much smaller memory footprint size, it is more efficient and can run in a cheap uniprocessor machine with moderate memory size as used in our experiments.

Other frequent sequence mining algorithms such as PrefixSpan[46] can also find long frequent sequences. Compared with these frequent sequence mining algorithms, *C-Miner* is more efficient for discovering long frequent sequences because it not only avoids searching the non-frequent sequences while generating longer sequences, but also prunes all the unpromising searching branches according to the closed sequence property as we have discussed. *C-Miner* can outperform PrefixSpan by an order of magnitude for some datasets.

4 Case Studies

4.1 Correlation-Directed Prefetching (CDP)

The block correlation information inferred by *C-Miner* can be used to prefetch more intelligently. Assume that *C-Miner* has obtained a block correlation rule: if block b_1 is accessed, block b_2 will also be accessed soon within a short distance (of length gap) with a certain confidence (probability). Based on this rule, when there is an access to block b_1, we can prefetch block b_2 into the storage cache since it will probably be accessed soon. Doing such can avoid future accesses to disks to fetch these blocks.

Several design issues should be considered while using block correlations for prefetching. One of the most important issues is how to effectively share the limited size cache for both caching and prefetching. If prefetching is too aggressive, it can pollute the storage cache and may even degrade the cache hit ratio and system performance. This problem has been investigated thoroughly by previous work [9, 10, 45]. We therefore do not investigate it further in our paper. In our simulation experiments, we simply fix the cache size for prefetched data so it does not compete with non-prefetching requests. However, the total cache size is fixed at the same value for the system with and without prefetching in order to have a fair comparison.

Another design issue is the extra disk load imposed by prefetch requests. If the disk load is too heavy, the disk utilization is close to 100%. In this case, prefetching can add significant overheads to demand requests, canceling out the benefits of improved storage cache hit ratio. Two methods can be used to alleviate this problem. The first method is to differentiate between demand requests and prefetch requests by using a priority-based disk scheduling scheme. In particular, the system uses two waiting queues in the disk scheduler: critical and non-critical. All the demand requests are issued to the critical queue, while the prefetch requests are issued to the non-critical queue which has lower priority.

The other method is to throttle the prefetch requests to a disk if the disk is heavily utilized. Since the correlation rules have different confidences, we can set a confidence threshold to limit the number of rules that are used for prefetching. All the rules with confidence lower than the threshold are ignored. Obviously, the higher the confidence threshold is, the fewer the rules are used. Therefore, CDP acts less aggressively. In order to adjust the threshold to make prefetching adapt to the current disk workload, we keep track of the current load on each disk. When the workload is too high, say the disk utilization is more than 80%, we increase the confidence threshold for correlation rules that direct the issuing of prefetch requests to this disk.

Once the disk load drops down to a low level, say the utilization is less than 50%, we decrease the confidence threshold for correlation rules so that more rules can be used for prefetching. By doing this, the overhead on disk bandwidth caused by prefetches is kept within an acceptable range.

4.2 Correlation-directed Disk Layout

Block correlations can help lay out data on disks to improve performance. The dominant latencies in a disk access are the seek time and rotation delay. So if correlated blocks can be allocated together on a disk and can be fetched using one disk access, the total seek time and rotation delay for all these blocks can be reduced. Thereafter, both of the throughput and the response time can be improved. But CDP is more effective than disk layout for improving response time as shown in section 5.5.

We can lay out the blocks on disks based on block correlatoins like that: if we know a correlation *abcd* from *C-Miner*, we can try to allocate them contiguously in a disk. Whenever any one of these blocks is read, all four blocks are fetched together into the storage cache using one disk access. Since some blocks may appear in several patterns, we allocate the block based on the rules with highest support value.

One of the main design issues is how to maintain the directory information and reorganize data without an impact on the foreground workload. After reorganizing disk layouts, we need to map logical block numbers to new physical block numbers. The mapping table might become very large. Some previous work has studied these issues and shown that disk layout reorganization is feasible to implement [50]. They proposed a two-tiered software architecture to combine multiple disk layout heuristics so that it adapts to different environments. Block correlation-directed disk layout can be one of the heuristics in their framework. Due to space limitation, we do not discuss this issue further.

5 Simulation Results

5.1 Evaluation Methodology

To evaluate the benefits of exploiting block correlations in block prefetching and disk data layout, we use trace-driven simulations with several large disk traces collected in real systems. Our simulator combines the widely used DiskSim simulator [20] with a storage cache simulator, CacheSim, to simulate a complete storage system. CacheSim implements the Least Recently Used (LRU) replacement policy. Accesses to the simulated storage system first go through a storage cache and only read misses or writes access physical disks. The simulated disk specification is similar to that of the 10000 RPM IBM Ultrastar 36Z15. The parameters are taken from the disk's data sheet [28, 11].

Our experiments use the following four real system traces:

- *TPC-C Trace* is an I/O trace collected on a storage system connected to a Microsoft SQL Server via storage area network. The Microsoft Server SQL clients connect to the Microsoft SQL Server via Ethernet and run the TPC-C benchmark [39] for 2 hours. The database consists of 256 warehouses and the footprint is 60GB, and the storage system employs a RAID of 4 disks. A more detailed description of this trace can be found in [69, 13].

- *Cello-92* was collected at Hewlett-Packard Laboratories in 1992 [49, 48]. It captured all low-level disk I/O performed by the system. We used the trace gathered on Cello, which is a timesharing system used by a group of researchers at HP Labs to do simulations, compilation, editing and email. The trace includes the accesses to 8 disks. We have also tried other HP disk trace files, and the results are similar.

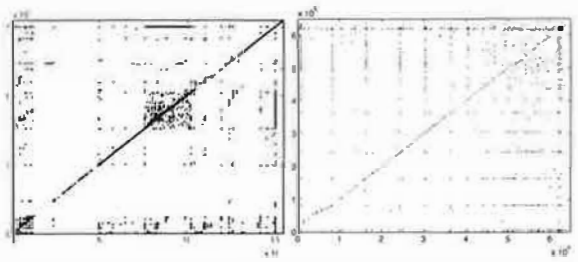
- *Cello-96* is similar to Cello-92. The only difference is that this trace was collected in 1996 and thereby contains more modern workloads. It includes the accesses to 20 disks from multiple users and miscellaneous applications. It contains a lot of sequential access patterns, so the simple sequential prefetching approaches can significantly benefit from them.

- *OLTP* is a trace of an OLTP application running at a large financial institution. It was made available by the Storage Performance Council [58]. The disk subsystem is composed of 19 disks.

All the traces are collected after filtering through a first-level buffer cache such as the database server cache. Fortunately, unlike other access patterns such temporal locality that can be filtered by large first-level buffer caches, most block correlations can still be discovered at the second-level. Only those correlations involving "hot" blocks that always stay at the first-level can be lost at the second-level. However, these correlations are not useful to exploit anyway since "hot" blocks are kept at the first-level and therefore are rarely accessed at the second-level.

In our experiments, we use only the first half part of the trace to mine block correlations using *C-Miner*. Using these correlation rules, we evaluate the performance of correlation-directed prefetching and data layout using the rest of the traces. The correlation rules are kept unchanged during the evaluation phase. For example, in Cello-92, we use the first 3-days' trace to mine block correlations and use the following 4 days to evaluate the correlation-directed prefetching and data layout. The reason for doing this is to show the stable characteristic of block correlations and predictive powers of our method.

To provide a more fair comparison, we also implement the commonly used sequential prefetching scheme. At non-consecutive misses to disks, the system also issues a prefetch request to load 16 consecutive blocks. We have also tried prefetching more or fewer blocks, but the results are similar or worse.



(a) *Cello-96 trace* (b) *TPC-C trace*

Figure 4: Block correlations mined from traces

5.2 Visualization of Block Correlations

5.2.1 Correlations in Real System Traces

Figure 4 plots the block correlations discovered by *C-Miner* from the Cello-96 and TPC-C traces. Since multi-block correlations are difficult to visualize, we plot only dual-block correlations. If there is an association rule $x \dashrightarrow y$, we plot a corresponding point at (x, y). Therefore, each point (x, y) in the graphs indicates a correlation between blocks x and y. Since the traces contain multiple disks' accesses, we plot the disk block address using a unified continuous address space by plotting one disk address space after another.

Simple patterns such as spatial locality can be demonstrated in such a correlation graph. It is indicated by dark areas around the diagonal line. This is because the spatial locality can be represented by an association rule $x \dashrightarrow (x \pm k)$ where k is a small number, which means that if block x accessed, its neighbor blocks are likely to be accessed soon. Since k is small, the points $(x, x \pm k)$ are around the diagonal line, as shown on the Cello-96 traces (Figure 4a). The graph for the TPC-C trace does not have such apparent characteristic, indicating TPC-C does not have strong spatial locality.

Some more complex patterns can also be seen from correlation graphs. For example, in figure 4b, there are many horizontal or vertical lines, indicating some blocks are correlated to many other blocks. Because this is a database I/O trace, these hot blocks with many correlations are likely to be the root of trees or subtrees. In the next subsection, we visualize block correlations specifically for tree structures.

5.2.2 Correlations in B-tree

In order to demonstrate the capability of *C-Miner* to discover semantics in a tree structure, we use a synthetic trace that simulates a client that searches data in a B-tree data structure, which is commonly used in databases. The B-tree maintains the indices for 5000 data items, each block has space for four search-key values and five pointers. We perform 1000 searches. To simulate a real-world situation where some "hot" data items are searched more frequently than others, searches are not uniformly distributed. Instead, we use a Zipf distribution and 80% of searches are to 100 "hot" data items.

The block correlations mined from the B-tree trace are visualized in figures 5. Note here constructing this tree does not take any semantic information from the application (the synthetic trace generator). The edges between nodes are reconstructed purely based on block correlations. Due to the space limitation, we only show part of the correlations. Each rule $x \dashrightarrow y$ is denoted as a directed edge with support as its weight. The figure illustrates that the block correlations implicate a tree-like structure. Also note that our approach to obtain block correlations is fully transparent without any assumption on storage front-ends.

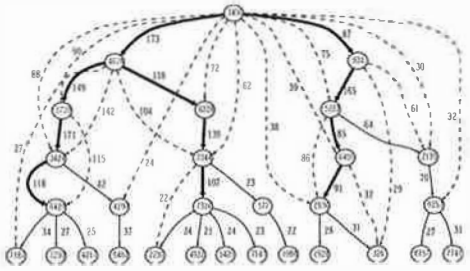


Figure 5: Block correlations in B-tree. The number on an edge is the support value for the corresponding correlation. The dashed lines indicate the correlation between a node and its descendants other than its children. The highlighted lines are the correlations with $support \geq 80$. Note that correlations with $support < 20$ are not produced by *C-Miner* ($min\ sup = 20$) in order to make the tree reasonably small and sparse for plotting.

5.3 Stability of Block Correlations

In order to show that block correlations are relatively stable, we use the correlation rules mined from the *first 3 days* of the Cello-92 trace. Our simulator applies these rules to the next 4 days' trace without updating any rules. Figure 6 shows the miss ratio for the *next 4 days'* trace using correlated-directed prefetching (CDP). The miss ratios in the figure are calculated by aggregating every 10000 read operations. This figure shows that CDP is always better than the base case. This implies that correlations mined from the first 3 days are still effective for the next 4 days. In other words, block correlations are relatively stable for a relative long period of time. Therefore, there is no need to run *C-Miner* continuously in the background to update block correlations. This also shows that, as long as the mining algorithm is reasonably efficient, the mining overhead is not a big issue.

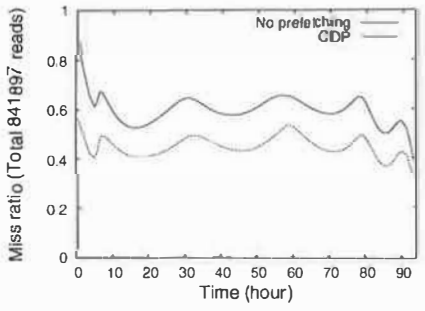


Figure 6: Miss ratio for Cello-92 (64MB; 4MB)

5.4 Data Mining Overhead

Table 1 shows the running time and space overheads for mining different traces. *C-Miner* is running on an Intel Xeon 2.4GHz machine and Windows 2000 Server. The time and space overhead does not depend on the confidence of rules as we discussed in section 3.2 but the number of rules does. The results show that *C-Miner* can effectively and practically discover block correlations for different workloads. For example, it takes less than 1 hour to discover half a million association rules from the Cello-96 trace that contains a full-day's disk requests. For the TPC-C trace, although it takes about 1 hour to mine 1 hour's trace, it is still practical for storage systems. Because block correlations are relatively stable, it is unnecessary to keep mining for correlations in the background. Instead, it might be acceptable to spend one hour every week on running *C-Miner* to update correlation rules. In our experiments, we only use parts of the traces to mine correlations, and use the remaining traces to evaluate correlation-directed prefetching and disk layout. Our experimental results indicate that correlations are relative stable and are useful for accesses made much later after the training period.

Training Trace	# of rules (K)	time (sec)	space (MB)
Cello-92 (3 days)	228	7800	3.1
Cello-96 (1 day)	514	2089	4.6
TPC-C (1 hour)	235	3355	9.2
OLTP (2.5 hours)	186	174	172.6

Table 1: Mining Overhead ($confidence \geq 10\%$)

C-Miner is also efficient in terms of space overhead for most of traces. It takes less than 10 MB to mine the Cello and TPC-C traces. With such a small requirement, the data mining can run on the same machine as the storage system without causing too much memory overhead. A uniprocessor PC with 512MB memory would do the work. In the future, we will investigate using stream mining algorithms to effectively mine continuous information with much less time and space overhead. The stream mining algorithm could be embedded in the storage controllers using their spare CPU power.

5.5 Correlation-directed Prefetching and Disk Layout

The bar graphs in figures 7a-d compare the read miss ratios and response times using the four different schemes: base-line (no-prefetching), sequential prefetching, correlation-directed prefetching (CDP), and correlation-directed prefetching and disk layout (CDP+layout). For the last three schemes with prefetching, the prefetch cache size is set to be the same. All four settings use the same total size of storage cache in order to make a fair comparison. In other words, the $TotalCacheSize$, which equals to the sum of $DemandCacheSize$ and $PrefetchCacheSize$, is the same for all four schemes.

CDP can improve the average I/O response time for the base-line case by up to 25%. For instance, in Cello-92, CDP has 24.4% lower storage cache hit ratios than the base-line case. This translates into 24.75% improvement in the average I/O response time. These improvements are due to the fact that prefetching reduces the number of capacity misses as well as the number of cold misses. When the cache size is small, some blocks are evicted and need to be fetched again for disks upon subsequent accesses. Prefetching can avoid misses at some of these accesses.

The improvement by CDP is much more significant than that by the commonly used sequential prefetching scheme, especially in the case of TPC-C and Cello-92. For example, for the TPC-C trace, sequential prefetching only slightly reduces the cache miss ratio (by only 2%), which is then completely cancelled out by the prefetching overheads. Therefore, sequential prefetching has even worse response time than the base case. For the other two traces (Cello-92 and OLTP trace), the improvement of the sequential prefetching scheme is very small, almost invisible in terms of the average response time. However, in Cello-96, sequential prefetching has a lower miss ratio and slightly better response time than CDP. This is because this trace has a lot of sequential accesses. But these sequential accesses are not frequent enough in the access stream so it is not caught by *C-Miner*. Fortunately, our patterns obtained by *C-Miner* can be complementary and combined with the existing online sequential prefetching algorithms that can detect non-frequent sequential access patterns.

CDP+layout has only small improvement over CDP. Obviously, CDP+layout should not affect cache miss ratio at all. It only matters to average I/O response time when the disk is heavily utilized. Its benefits are only visible in the Cello-96 trace, where CDP+layout has 4% better average response time than CDP. This small improvement indicates that our optimization for hiding prefetching overheads using priority-based disk scheduling is already good enough. Therefore, disk layout does not provide significant benefits. However, when the disk is too heavily utilized for the disk scheduling scheme to hide most of the prefetching overheads, we expect the benefit of correlation-directed disk layout will be larger.

5.6 Impact of Configurations

5.6.1 *Effects of the Confidence Threshold*

A parameter that might affect the benefits of correlation directed prefetching is the confidence threshold. Figure 8 shows the effects of varying the confidence threshold from 0% to 90% . A lower confidence threshold corresponds to a more aggressive prefetching policy. The figure shows that the miss ratio is minimum when prefetching is most aggressive and all the rules are used. We can see that the line is flat when the confidence is smaller than 30%, which indicates that the rules with small confidence are useless.

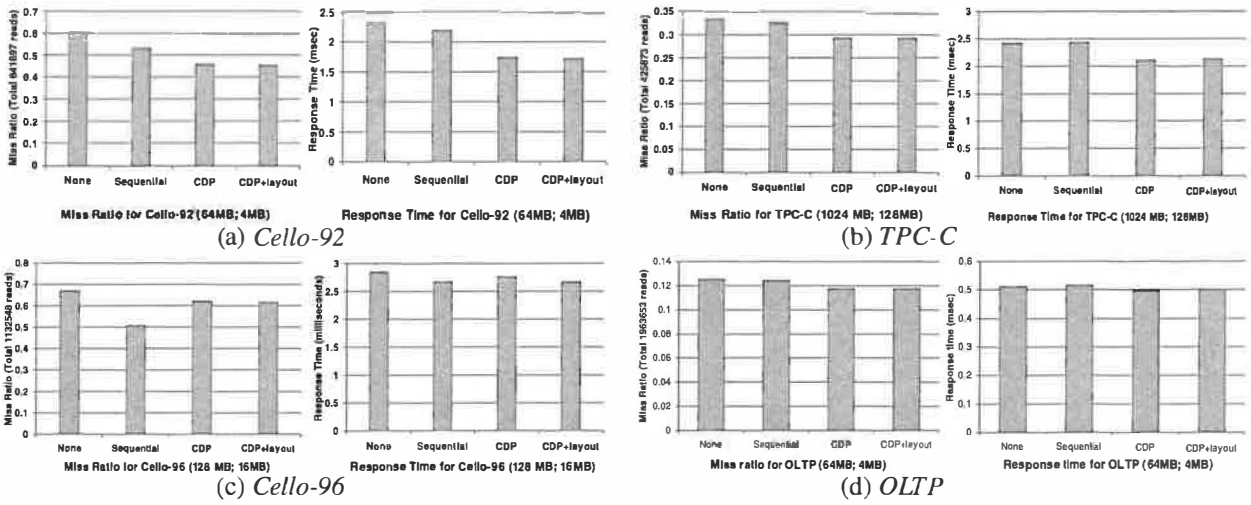


Figure 7: Miss Ratio and Response Time. The first number in the parenthesis is the total cache size, and the second number is the prefetch cache size. In the base-line case "None", the prefetch cache size is 0, so the demand cache size = the total cache size. In the other three schemes, the demand cache size is the total cache size − the prefetch cache size.

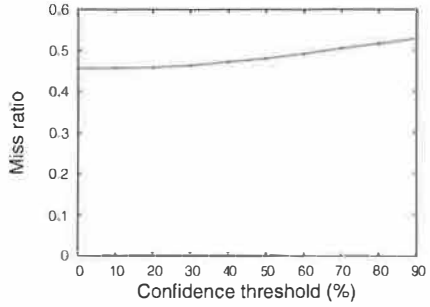


Figure 8: Effect of the confidence threshold (Cello-92)

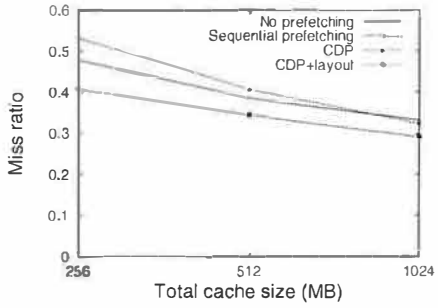


Figure 9: Effect of varying the total cache size for TPC-C

5.6.2 *Effects of the Total Cache Sizes*

If the cache size is comparable to the footprint of a trace, the system simply caches all accesses. Because of this, the read misses in the case of no prefetching are predominantly cold misses since subsequent accesses will be cache hits and will not go to the disk. Therefore, the prefetching schemes do not yield much improvement in performance.

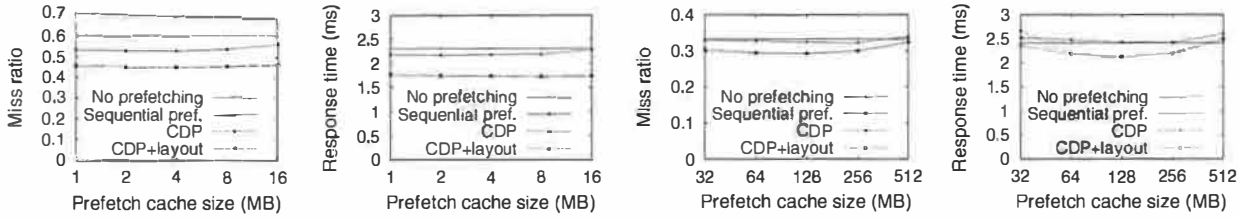
We study the effects of the cache size by varying the cache size for TPC-C exponentially from 256 MB to 1024 MB. In this experiment, we keep the prefetch cache fixed at 128 MB and vary the size of the demand cache. As expected, when the cache size is set at 256 MB, CDP+layout shows an improvement of 14.62% in miss ratio while with 512 MB, the improvement is only 10.58%. It is important to note that our workloads have relatively small working set sizes. In large real systems, it is usually not the case that the entire working set can fit into main memory.

5.6.3 *Effects of the Prefetch Cache Size*

Figures 10a-d show the effects of varying the prefetch cache size while the total cache size is fixed. In TPC-C, for instance, the storage cache miss ratio with CDP initially decreases as the prefetch cache size increases. It reaches the minimum when the prefetch cache size is set to 128 MB. Beyond that point the miss ratio increases again with the prefetch cache size. This phenomenon is also true for the sequential prefetching, even though it performs worse than CDP with almost all prefetch cache sizes.

The above phenomenon can be quite expected. When the prefetch cache size is very small, prefetched blocks may be replaced even before they are used. Even though the demand cache size is increased correspondingly, its benefit is not large enough to offset the loss in unused prefetches. Even worse, the overhead imposed by CDP causes an increase in the response time compared to the base case, as shown on Figure 10d. Fortunately, in this case, the CDP+layout starts to show the benefits of correlation-directed disk layout. It is still able to provide some small improvement over the base line case.

As the prefetch cache size increases, blocks can be retained longer in the prefetch cache and subsequently be used to handle block requests. However, the increase in prefetch cache size corresponds to a reduced demand cache size, but the benefit of prefetching in reducing misses outweighs the loss in the demand cache. Beyond 128 MB, increasing the prefetch cache size no longer has benefits for increasing hit ratio. So the loss due to the reduced demand cache size starts to dominate. Therefore, the overall miss ratio increases.



(a) *Cello-92 - miss ratio* (b) *Cello-92 - response time* (c) *TPC-C - miss ratio* (d) *TPC-C - response time*

Figure 10: Effect of varying the prefetch cache size (Note: the total cache size is fixed. Therefore, the demand cache size is also changing with the prefetch cache size. In the base-line case, there is no prefetch cache)

6 Related Work

In this section, we briefly discuss some representative work that is closely related to our work. Section 2 has discussed various approaches to capture data semantics. Thus, we do not repeat them here.

Data prefetching has also been studied extensively in databases, file systems and parallel applications with intensive I/Os. Most of previous prefetching work either relies on applications to pass hints or is based on simple heuristics such as sequential accesses. Examples of prefetching studies for databases include [56, 63, 44, 22, 21] as well some recent work [53] for mobile data delivery environments. Prefetching for file I/Os include application-controlled prefetching [9, 10] and informed prefetching [60, 32, 45], just to name a few. [57] is an example of prefetching in disk caches. I/O prefetching for out-of-core applications include compiler-assisted prefetching [43, 8] and prefetching through speculative execution [12].

In the spectrum of sophisticated prefetching schemes, research has been conducted for semantic distance-based file prefetching for mobile or networked file servers. Besides the probability graph-based approach described in Section 2, the SEER project from UCLA [34, 35] groups related files into clusters by keeping track of semantic distances between files and downloading as many complete clusters as possible onto the mobile station. The CLUMP project tries to leverage the concept of semantic distance to prefetch file clusters [18]. Kroeger extends the probability graph to a trie with each node representing the sequence of consecutive file accesses from the root to the node [33]. Lei and Duchamp also use a similar structure by building a probability tree [59, 38]. Vellanki and Chervenak combine Patterson's cost-benefit analysis with probabilistic prefetching for high performance parallel file systems [61]. Similar to the probability graph, most of these approaches may be feasible for prefetching at file granularity, but are impractical to track block correlations in a storage system (see Section 2).

Some studies used data compression techniques for prefetching. It was first proposed by Vitter and Krishnan [62]. The basic idea is to encode the data expected with higher probability using fewer bits. The prefetchers based on any optimal character-by-character data compressor were theoretically proven to be optimal in page fault rate. Later, [17] analyzed some practical issues of such a technique, and proposed three practical data compressors for prefetching.

Data mining methods have been mostly used to discover patterns in sales, finance or bio-informatics databases [27, 26]. Only a few studies have applied them in systems. A well-known example is using data mining for intrusion detection [37, 16]. Data mining has recently been used in performance evaluation [40] to model bursty traffic.

Data mining and machine learning have been used in web environments to predict HTTP requests. Schechter et al. introduced path profiling to predict HTTP requests in web environments [51]. Pitkow and Pirolli have used longest repeating subsequences to perform path matching for predicting web accesses from a client [47]. These schemes predict the next HTTP request by matching the surfer's current sequence against the path profile database.

While path-based prediction may work very well for web environments, it is very difficult to capture block correlations in storage systems. This is because web browser/server workloads are different from storage workloads. Each web client usually only browses one page at a time, whereas a storage front-end such as database server can have hundreds of outstanding requests. Since the path-matching schemes do not allow any gaps in the subsequence or path, they cannot be used easily to capture block correlations in a storage system. To support gaps or lookahead distances in these work will suffer the same problem as the probability graph-based approach.

Our work is also related to various adaptive approaches using learning techniques [41, 4], intelligent storage cache management [69, 65, 42, 13], and autonomic storage systems [64, 2, 30]

7 Conclusions and Future Work

This paper proposes *C-Miner*, a novel approach that uses data mining techniques to systematically mine access sequences in a storage system to infer block correlations. More specifically, we have designed a frequent sequence mining algorithm to find correlations among blocks. Using several large real system disk traces, our experiments show that *C-Miner* is reasonably fast with small space requirement and is therefore practical to be used on-line in an autonomic storage system. We have also evaluated correlation-directed prefetching and data layout. Our experimental results with real-system traces have shown that correlation-directed prefetching and data layout can improve I/O average response time by 12-25% compared to no-prefetching, and 7-20% compared to the commonly used sequential prefetching.

Our study has several limitations. First, even though this paper focuses on how to obtain block correlations, our evaluation of the block correlation-directed prefetching and disk layout was conducted using only simulations. We are in the process of implementing correlation-directed prefetching and disk layout in our previously built storage system. Second, we have only evaluated four real-system workloads, it would be interesting to evaluate other workloads such as those with substantially sequential accesses. Third, we do not compare with the semantic-distance graph approach. The main reason is that our preliminary experiment indicates that the SD graphs significantly exceed the memory space, making it extremely slow and almost infeasible to build such graphs.

Currently, *C-Miner* does not consider the recency factor, that is, some recent frequent subsequences may be more important than other "ancient" frequent subsequences. To mine time-series data, our algorithm needs to be modified to change the support or confidence value of a rule dynamically as time progresses. Currently, we are designing efficient stream data mining algorithms specifically for mining access sequences for storage systems and any other similar situations.

8 Acknowledgements

The authors would like to thank the shepherd, Jeff Chase, and the anonymous reviewers for their invaluable feedback. We appreciate Kimberly Keeton of HP labs for the constructive discussion and thank HP storage system labs for providing us cello traces. We are also grateful to Professor Jiawei Han and his student Xifeng Yan for their help with the CloSpan algorithm and insightful discussions. This research is supported by the NSF CCR-0305854 grant and IBM CAS Fellowship. Our experiments were conducted on equipment provided through the IBM SUR grant.

REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Eleventh International Conference on Data Engineering*, 1995.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, 2002.

[3] G. H. Anthes. Storage virtualization: The next step. Computerworld, January 28, 2002.

[4] I. Ari, A. Amer, E. Miller, S. Brandt, and D. Long. Who is more adaptive? ACME: adaptive caching using multiple experts. In Workshop on Distributed Data and Structures (WDAS), 2002.

[5] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[6] J. Ayres, J. E. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using bitmaps. In *Proc. 2002 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'02)*, pages 429–435, Edmonton, Canada, July 2002.

[7] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 3–14. IEEE Press, 1998.

[8] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2):111–170, 2001.

[9] P. Cao, E. Felten, and K. Li. Application-controlled file caching policies. In *USENIX Summer 1994 Technical Conference*, pages 171–182, June 1994.

[10] P. Cao, E. W. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of ACM SIGMETRICS*, May 1995.

[11] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th International Conference on Supercomputing*, June 2003.

[12] F. W. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Operating Systems Design and Implementation*, pages 1–14, 1999.

[13] Z. Chen, Y. Zhou, and K. Li. Eviction-based cache placement for storage caches. In *USENIX Annual Technical Conference, General Track*, pages 269–281, San Antonio, Texas, USA, June 9-14 2003.

[14] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2000.

[15] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 127–141, Dublin, Ireland, 1993.

[16] C. Clifton and G. Gengo. Developing custom intrusion detection filters using data mining. In *2000 Military Communications International Symposium (MILCOM2000)*, Los Angeles, California, Oct. 2000.

[17] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proc. 1993 ACM-SIGMOD Conference on Management of Data*, pages 257–266, May 1993.

[18] P. R. Eaton, D. Geels, and G. Mori. Clump: Improving file system performance through adaptive optimizations.

[19] EMC Corporation. Symmetrix 3000 and 5000 Enterprise Storage Systems product description guide., 1999.

[20] G. Ganger. Systemoriented evaluation of I/O subsystem performance. Technical Report CSE-TR-243-95, University of Michigan, June 1995.

[21] C. A. Gerlhof and A. Kemper. A multi-threaded architecture for prefetching in object bases. In M. Jarke, J. A. B. Jr., and K. G. Jeffery, editors, *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, volume 779 of *Lecture Notes in Computer Science*, pages 351–364. Springer, 1994.

[22] C. A. Gerlhof and A. Kemper. Prefetch support relations in object bases. In M. P. Atkinson, D. Maier, and V. Benzaken, editors, *Persistent Object Systems, Proceedings of the Sixth International Workshop on Persistent Object Systems, Tarascon, Provence, France, 5-9 September 1994*, Workshops in Computing, pages 115–126. Springer and British Computer Society, 1994.

[23] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[24] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *In Proceedings of the 1994 Summer USENIX Conference*, 1994.

[25] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *In Proceedings of the International Conference on Parallel and Distributed Computing Systems*, 1995.

[26] J. Han. How can data mining help bio-data analysis? In *Proc. 2002 Workshop on Data Mining in Bioinformatics (BIOKDD'02)*, pages 1–4, Edmonton, Canada, July 2002.

[27] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[28] IBM hard disk drive - Ultrastar 36Z15.

[29] IBM. Storage Tank, a distributed storage system. IBM White Paper, http://www.almaden.ibm.com/StorageSystems/file_ systems/ storage tank/papers.shtml.

[30] K. Keeton and J. Wilkes. Automating data dependability. Proceedings of the 10th ACM-SIGOPS European Workshop, 2002.

[31] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 119–134, San Diego, CA, Oct. 2000.

[32] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. R. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 19–34. USENIX Association, 1996.

[33] T. M. Kroeger and D. D. E. Long. Predicting file-system actions from prior events. In *1996 USENIX Annual Technical Conference*, pages 319–328, 1996.

[34] G. Kuenning. Design of the SEER predictive caching scheme. In *Workshop on Mobile Computing Systems and Applications*, 1994.

[35] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 264–275, St. Malo, France, Oct. 1997. ACM.

[36] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 84–92. ACM Press, 1996.

[37] W. Lee and S. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.

[38] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference*, Anaheim, California, USA, 1997.

[39] S. T. Leutenegger and D. Dias. A modeling study of the TPC-C benchmark. *SIGMOD Record*, 22(2):22–31, June 1993.

[40] T. M. M. Wang, N. Chan, S. Papadimitriou, and C. Faloutsos. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. 18th Internal Conference on Data Engineering, 2002.

[41] T. M. Madhyastha, G. A. Gibson, and C. Faloutsos. Informed prefetching of collective input/output requests. Proceedings of SC99.

[42] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST 03), San Franciso, CA*, 2003.

[43] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, Oct. 1996.

[44] M. Palmer and S. B. Zdonik. Fido: A cache that learns to fetch. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 255–264. Morgan Kaufmann, 1991.

[45] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *the 15th ACM Symposium on Operating System Principles*, 1995.

[46] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215–224, Heidelberg, Germany, April 2001.

[47] J. E. Pitkow and P. Pirolli. Mining longest repeating subsequences to predict world wide web surfing. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[48] C. Ruemmler and J. Wilkes. A trace-driven analysis of disk working set sizes. Technical Report HPL–OSR–93–23, Hewlett-Packard Laboratories, Palo Alto, CA, USA, Apr. 5 1993.

[49] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Proceedings of the Winter 1993 USENIX Conference*, 1993.

[50] B. Salmon, E. Thereska, C. A. Soules, and G. R. Ganger. A two-tiered software architecture for automated tuning of disk layouts. In *First Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[51] S. Schechter, M. Krishnan, and M. D. Smith. Using path profiles to predict http requests. In *Seventh Intl World Wide Web Conference*, Apr 1998.

[52] J. Schindler, J. Griffin, C. Lumb, and G. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, 2002.

[53] A. Seifert and M. H. Scholl. A multi-version cache replacement and prefetching policy for hybrid data delivery environments. In *28th International Conference on Very Large Data Bases (VLDB)*, 2002.

[54] M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies*, 2003.

[55] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, Sept. 1978.

[56] B. J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings of International Conference on Parallel Proc essing*, pages 6–8, 1978.

[57] V. Soloviev. Prefetching in segmented disk cache for multi-disk systems. In *Proceedings of the fourth workshop on I/O in parallel and distributed systems*, pages 69–82. ACM Press, 1996.

[58] Storage Performance Council. SPC I/O traces. http://www.storageperformance.org/.

[59] C. D. Tait, H. Lei, S. Acharya, and H. Chang. Intelligent file hoarding for mobile computers. In *Mobile Computing and Networking*, pages 119–125, 1995.

[60] A. Tomkins, R. H. Patterson, and G. Gibson. Informed multi-process prefetching and caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 100–114. ACM Press, 1997.

[61] V. Vellanki and A. Chervenak. A cost-benefit scheme for high performance predictive prefetching. In *Proceedings of SC99: High Performance Networking and Computing*, Portland, OR, 1999. ACM Press and IEEE Computer Society Press.

[62] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, Oct 1991.

[63] H. Wedekind and G. Zoerntlein. Prefetching in realtime database applications. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 215–226. ACM Press, 1986.

[64] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. Proc. of the 15th Symp. on Operating Systems Principles, 1995.

[65] T. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *USENIX*, 2002.

[66] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *Proc. 2003 SIAM Int. Conf. Data Mining (SDM'03)*, San Fransisco, CA, May 2003.

[67] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.

[68] Y. Zhang, J. Zhang, A. Sivasubramaniam, C. Liu, and H. Franke. Decision-support workload characteristics on clustered database server from the OS perspective. In *Proceedings of the International Conference on Distributed Conputing Systems(ICDCS)*, May 2003.

[69] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the Usenix Technical Conference*, June 2001.

CAR: Clock with Adaptive Replacement

Sorav Bansal[†] **and Dharmendra S. Modha**[‡]
[†]Stanford University, [‡]IBM Almaden Research Center
Emails: sbansal@stanford.edu, dmodha@us.ibm.com

***Abstract*—CLOCK** is a classical cache replacement policy dating back to 1968 that was proposed as a low-complexity approximation to **LRU**. On every cache hit, the policy **LRU** needs to move the accessed item to the most recently used position, at which point, to ensure consistency and correctness, it serializes cache hits behind a single global lock. **CLOCK** eliminates this *lock contention*, and, hence, can support high concurrency and high throughput environments such as virtual memory (for example, Multics, UNIX, BSD, AIX) and databases (for example, DB2). Unfortunately, **CLOCK** is still plagued by disadvantages of **LRU** such as disregard for "frequency", susceptibility to scans, and low performance.

As our main contribution, we propose a simple and elegant new algorithm, namely, **CLOCK** with Adaptive Replacement (**CAR**), that has several advantages over **CLOCK**: (i) it is scan-resistant; (ii) it is self-tuning and it adaptively and dynamically captures the "recency" and "frequency" features of a workload; (iii) it uses essentially the same primitives as **CLOCK**, and, hence, is low-complexity and amenable to a high-concurrency implementation; and (iv) it outperforms **CLOCK** across a wide-range of cache sizes and workloads. The algorithm **CAR** is inspired by the Adaptive Replacement Cache (**ARC**) algorithm, and inherits virtually all advantages of **ARC** including its high performance, but does not serialize cache hits behind a single global lock. As our second contribution, we introduce another novel algorithm, namely, CAR with Temporal filtering (**CART**), that has all the advantages of **CAR**, but, in addition, uses a certain temporal filter to distill pages with long-term utility from those with only short-term utility.

I. Introduction

A. Caching and Demand Paging

Modern computational infrastructure is rich in examples of memory hierarchies where a fast, but expensive main ("cache") memory is placed in front of a cheap, but slow auxiliary memory. Caching algorithms manage the contents of the cache so as to improve the overall performance. In particular, cache algorithms are of tremendous interest in databases (for example, DB2), virtual memory management in operating systems (for example, LINUX), storage systems (for example, IBM ESS, EMC Symmetrix, Hitachi Lightning), etc., where cache is RAM and the auxiliary memory is a disk subsystem.

In this paper, we study the generic cache replacement problem and will not concentrate on any specific application. For concreteness, we assume that both the cache and the auxiliary memory are managed in discrete, uniformly-sized units called "pages". If a requested page is present in the cache, then it can be served quickly resulting in a "cache hit". On the other hand, if a requested page is not present in the cache, then it must be fetched from the auxiliary memory resulting in a "cache miss". Usually, latency on a cache miss is significantly higher than that on a cache hit. Hence, caching algorithms focus on improving the hit ratio.

Historically, the assumption of "demand paging" has been used to study cache algorithms. Under demand paging, a page is brought in from the auxiliary memory to the cache only on a cache miss. In other words, demand paging precludes speculatively pre-fetching pages. Under demand paging, the only question of interest is: When the cache is full, and a new page must be inserted in the cache, which page should be replaced? The best, offline cache replacement policy is Belady's **MIN** that replaces the page that is used farthest in the future [1]. Of course, in practice, we are only interested in online cache replacement policies that do not demand any prior knowledge of the workload.

B. LRU: Advantages and Disadvantages

A popular online policy imitates **MIN** by replacing the least recently used (**LRU**) page. So far, **LRU** and its variants are amongst the most popular replacement policies [2], [3], [4]. The advantages of **LRU** are that it is extremely simple to implement, has constant time and space overhead, and captures "recency" or "clustered locality of reference" that is common to many workloads. In fact, under a certain Stack Depth Distribution (SDD) assumption for workloads, **LRU** is the optimal cache replacement policy [5].

The algorithm **LRU** has many disadvantages:

- D1 On every hit to a cache page it must be moved to the most recently used (**MRU**) position. In an asynchronous computing environment where multiple threads may be trying to move pages to the MRU position, the MRU position is protected by a lock to ensure consistency and correctness. This lock typically leads to a great amount of contention, since all cache hits are serialized behind this lock. Such contention is often unacceptable in high performance and high throughput environments such as virtual memory, databases, file systems, and storage controllers.

D2 In a virtual memory setting, the overhead of moving a page to the MRU position–on every page hit–is unacceptable [3].

D3 While LRU captures the "recency" features of a workload, it does not capture and exploit the "frequency" features of a workload [5, p. 282]. More generally, if some pages are often re-requested, but the temporal distance between consecutive requests is larger than the cache size, then LRU cannot take advantage of such pages with "long-term utility".

D4 LRU can be easily polluted by a scan, that is, by a sequence of one-time use only page requests leading to lower performance.

C. CLOCK

Frank Corbató (who later went on to win the ACM Turing Award) introduced CLOCK [6] as a one-bit approximation to LRU:

> "In the Multics system a paging algorithm has been developed that has the implementation ease and low overhead of the FIFO strategy and is an approximation to the LRU strategy. In fact, the algorithm can be viewed as a particular member of a class of algorithms which embody for each page a shift register memory length of k. At one limit of $k = 0$, the algorithm becomes FIFO; at the other limit as $k \to \infty$, the algorithm is LRU. The current Multics system is using the value of $k = 1$, ..."

CLOCK removes disadvantages D1 and D2 of LRU. The algorithm CLOCK maintains a "page reference bit" with every page. When a page is first brought into the cache, its page reference bit is set to zero. The pages in the cache are organized as a circular buffer known as a *clock*. On a hit to a page, its page reference bit is set to one. Replacement is done by moving a *clock hand* through the circular buffer. The clock hand can only replace a page with page reference bit set to zero. However, while the clock hand is traversing to find the victim page, if it encounters a page with page reference bit of one, then it resets the bit to zero. Since, on a page hit, there is no need to move the page to the MRU position, no serialization of hits occurs. Moreover, in virtual memory applications, the page reference bit can be turned on by the hardware. Furthermore, performance of CLOCK is usually quite comparable to LRU. For this reason, variants of CLOCK have been widely used in Multics [6], DB2 [7], BSD [8], AIX, and VAX/VMS [9]. The importance of CLOCK is further underscored by the fact that major textbooks on operating systems teach it [3], [4].

D. Adaptive Replacement Cache

A recent breakthrough generalization of LRU, namely, Adaptive Replacement Cache (ARC), removes disadvantages D3 and D4 of LRU [10], [11]. The algorithm ARC is scan-resistant, exploits both the recency and the frequency features of the workload in a self-tuning fashion, has low space and time complexity, and outperforms LRU across a wide range of workloads and cache sizes. Furthermore, ARC which is self-tuning has performance comparable to a number of recent, state-of-the-art policies even when these policies are allowed the best, offline values for their tunable parameters [10, Table V].

E. Our Contribution

To summarize, CLOCK removes disadvantages D1 and D2 of LRU, while ARC removes disadvantages D3 and D4 of LRU. In this paper, as our main contribution, we present a simple new algorithm, namely, Clock with Adaptive Replacement (CAR), that removes all four disadvantages D1, D2, D3, and D4 of LRU. The basic idea is to maintain two clocks, say, T_1 and T_2, where T_1 contains pages with "recency" or "short-term utility" and T_2 contains pages with "frequency" or "long-term utility". New pages are first inserted in T_1 and graduate to T_2 upon passing a certain test of long-term utility. By using a certain precise history mechanism that remembers recently evicted pages from T_1 and T_2, we adaptively determine the sizes of these lists in a data-driven fashion. Using extensive trace-driven simulations, we demonstrate that CAR has performance comparable to ARC, and substantially outperforms both LRU and CLOCK. Furthermore, like ARC, the algorithm CAR is self-tuning and requires no user-specified magic parameters.

The algorithms ARC and CAR consider two consecutive hits to a page as a test of its long-term utility. At upper levels of memory hierarchy, for example, virtual memory, databases, and file systems, we often observe two or more successive references to the same page fairly quickly. Such quick successive hits are not a guarantee of long-term utility of a pages. Inspired by the "locality filtering" principle in [12], we introduce another novel algorithm, namely, CAR with Temporal filtering (CART), that has all the advantages of CAR, but, imposes a more stringent test to demarcate between pages with long-term utility from those with only short-term utility.

We expect that CAR is more suitable for disk, RAID, storage controllers, whereas CART may be more suited to virtual memory, databases, and file systems.

F. Outline of the Paper

In Section II, we briefly review relevant prior art. In Sections III and IV, we present the new algorithms CAR and CART, respectively. In Section V, we present results of trace driven simulations. Finally, in Section VI, we present some discussions and conclusions.

II. Prior Work

For a detail bibliography of caching and paging work prior to 1990, see [13], [14].

A. LRU and LFU: Related Work

The Independent Reference Model (IRM) captures the notion of frequencies of page references. Under the IRM, the requests at different times are stochastically independent. LFU replaces the least frequently used page and is optimal under the IRM [5], [15] but has several drawbacks: (i) Its running time per request is logarithmic in the cache size. (ii) It is oblivious to recent history. (iii) It does not adapt well to variable access patterns; it accumulates stale pages with past high frequency counts, which may no longer be useful.

The last fifteen years have seen development of a number of novel caching algorithms that have attempted to combine "recency" (LRU) and "frequency" (LFU) with the intent of removing one or more disadvantages of LRU. Chronologically, FBR [12], LRU-2 [16], 2Q [17], LRFU [18], [19], MQ [20], and LIRS [21] have been proposed. For a detailed overview of these algorithms, see [19], [20], [10]. It turns out, however, that each of these algorithms leaves something to be desired, see [10]. The cache replacement policy ARC [10] seems to eliminate essentially all drawbacks of the above mentioned policies, is self-tuning, low overhead, scan-resistant, and has performance similar to or better than LRU, LFU, FBR, LRU-2, 2Q, MQ, LRFU, and LIRS–even when some of these policies are allowed to select the best, offline values for their tunable parameters–without any need for pre-tuning or user-specified magic parameters.

Finally, all of the above cited policies, including ARC, use LRU as the building block, and, hence, continue to suffer from drawbacks D1 and D2 of LRU.

B. CLOCK: Related Work

As already mentioned, the algorithm CLOCK was developed specifically for low-overhead, low-lock-contention environment.

Perhaps the oldest algorithm along these lines was First-In First-Out (FIFO) [3] that simply maintains a list of all pages in the cache such that *head* of the list is the oldest arrival and *tail* of the list is the most recent arrival. FIFO was used in DEC's VAX/VMS [9]; however, due to much lower performance than LRU, FIFO in its original form is seldom used today.

Second chance (SC) [3] is a simple, but extremely effective enhancement to FIFO, where a page reference bit is maintained with each page in the cache while maintaining the pages in a FIFO queue. When a page arrives in the cache, it is appended to the tail of the queue and its reference bit set to zero. Upon a page hit, the page reference bit is set to one. Whenever a page must be replaced, the policy examines the page at the head of the FIFO queue and replaces it if its page reference bit is zero otherwise the page is moved to the tail and its page reference bit is reset to zero. In the latter case, the replacement policy reexamines the new page at the head of the queue, until a replacement candidate with page reference bit of zero is found.

A key deficiency of SC is that it keeps moving pages from the head of the queue to the tail. This movement makes it somewhat inefficient. CLOCK is functionally identical to SC except that by using a circular queue instead of FIFO it eliminates the need to move a page from the head to the tail [3], [4], [6]. Besides its simplicity, the performance of CLOCK is quite comparable to LRU [22], [23], [24].

While CLOCK respects "recency", it does not take "frequency" into account. A generalized version, namely, GCLOCK, associates a counter with each page that is initialized to a certain value. On a page hit, the counter is incremented. On a page miss, the rotating clock hand sweeps through the clock decrementing counters until a page with a count of zero is found [24]. A analytical and empirical study of GCLOCK [25] showed that "its performance can be either better or worse than LRU". A fundamental disadvantage of GCLOCK is that it requires counter increment on every page hit which makes it infeasible for virtual memory.

There are several variants of CLOCK, for example, the two-handed clock [9], [26] is used by SUN's Solaris. Also, [6] considered multi-bit variants of CLOCK as finer approximations to LRU.

III. CAR

A. ARC: A Brief Review

Suppose that the cache can hold c pages. The policy ARC maintains a cache directory that contains $2c$ pages–c pages in the cache and c history pages. The cache directory of ARC, which was referred to as DBL in [10], maintains two lists: L_1 and L_2. The first list contains pages that have been seen only once recently, while the latter contains pages that have been seen at least twice recently. The list L_1 is thought of as "recency" and L_2 as "frequency". A more precise interpretation would have been to think of L_1 as "short-term utility" and L_2 as "long-term utility". The replacement policy

for managing DBL is: Replace the LRU page in L_1, if $|L_1| = c$; otherwise, replace the LRU page in L_2. The policy ARC builds on DBL by carefully selecting c pages from the $2c$ pages in DBL. The basic idea is to divide L_1 into top T_1 and bottom B_1 and to divide L_2 into top T_2 and bottom B_2. The pages in T_1 and T_2 are in the cache and in the cache directory, while the history pages in B_1 and B_2 are in the cache directory but not in the cache. The pages evicted from T_1 (resp. T_2) are put on the history list B_1 (resp. B_2). The algorithm sets a target size p for the list T_1. The replacement policy is simple: Replace the LRU page in T_1, if $|T_1| \geq p$; otherwise, replace the LRU page in T_2. The adaptation comes from the fact that the target size p is continuously varied in response to an observed workload. The adaptation rule is also simple: Increase p, if a hit in the history B_1 is observed; similarly, decrease p, if a hit in the history B_2 is observed. This completes our brief description of ARC.

B. CAR

Our policy CAR is inspired by ARC. Hence, for the sake of consistency, we have chosen to use the same notation as that in [10] so as to facilitate an easy comparison of similarities and differences between the two policies.

For a visual description of CAR, see Figure 1, and for a complete algorithmic specification, see Figure 2. We now explain the intuition behind the algorithm.

For concreteness, let c denote the cache size in pages. The policy CAR maintains four doubly linked lists: T_1, T_2, B_1, and B_2. The lists T_1 and T_2 contain the pages in cache, while the lists B_1 and B_2 maintain history information about the recently evicted pages. For each page in the cache, that is, in T_1 or T_2, we will maintain a page reference bit that can be set to either one or zero. Let T_1^0 denote the pages in T_1 with a page reference bit of zero and let T_1^1 denote the pages in T_1 with a page reference bit of one. The lists T_1^0 and T_1^1 are introduced for expository reasons only–they will not be required explicitly in our algorithm. Not maintaining either of these lists or their sizes was a key insight that allowed us to simplify ARC to CAR.

The precise definition of the four lists is as follows.

Each page in T_1^0 and each history page in B_1 has either been requested exactly once since its most recent removal from $T_1 \cup T_2 \cup B_1 \cup B_2$ or it was requested only once (since inception) and was never removed from $T_1 \cup T_2 \cup B_1 \cup B_2$.

Each page in T_1^1, each page in T_2, and each history page in B_2 has either been requested more than once since its most recent removal from $T_1 \cup T_2 \cup B_1 \cup B_2$, or was requested more than once and was never removed from $T_1 \cup T_2 \cup B_1 \cup B_2$.

Intuitively, $T_1^0 \cup B_1$ contains pages that have been seen exactly once recently whereas $T_1^1 \cup T_2 \cup B_2$ contains pages that have been seen at least twice recently. We roughly think of $T_1^0 \cup B_1$ as "recency" or "short-term utility" and $T_1^1 \cup T_2 \cup B_2$ as "frequency" or "long-term utility".

In the algorithm in Figure 2, for a more transparent exposition, we will think of the lists T_1 and T_2 as second chance lists. However, SC and CLOCK are the same algorithm that have slightly different implementations. So, in an actual implementation, the reader may wish to use CLOCK so as to reduce the overhead somewhat. Figure 1 depicts T_1 and T_2 as CLOCKs. The policy ARC employs a strict LRU ordering on the lists T_1 and T_2 whereas CAR uses a one-bit approximation to LRU, that is, SC. The lists B_1 and B_2 are simple LRU lists.

We impose the following invariants on these lists:

I1 $0 \leq |T_1| + |T_2| \leq c$.
I2 $0 \leq |T_1| + |B_1| \leq c$.
I3 $0 \leq |T_2| + |B_2| \leq 2c$.
I4 $0 \leq |T_1| + |T_2| + |B_1| + |B_2| \leq 2c$.
I5 If $|T_1| + |T_2| < c$, then $B_1 \cup B_2$ is empty.
I6 If $|T_1| + |B_1| + |T_2| + |B_2| \geq c$, then $|T_1| + |T_2| = c$.
I7 Due to demand paging, once the cache is full, it remains full from then on.

The idea of maintaining extra history pages is not new, see, for example, [16], [17], [19], [20], [21], [10]. We will use the extra history information contained in lists B_1 and B_2 to guide a continual adaptive process that keeps readjusting the sizes of the lists T_1 and T_2. For this purpose, we will maintain a *target size* p for the list T_1. By implication, the target size for the list T_2 will be $c - p$. The extra history leads to a negligible space overhead.

The list T_1 may contain pages that are marked either one or zero. Suppose we start scanning the list T_1 from the head towards the tail, until a page marked as zero is encountered; let T_1' denote all the pages seen by such a scan, until a page with a page reference bit of zero is encountered. The list T_1' does not need to be constructed, it is defined with the sole goal of stating our cache replacement policy.

The cache replacement policy CAR is simple:

> If $T_1 \setminus T_1'$ contains p or more pages, then remove a page from T_1, else remove a page from $T_1' \cup T_2$.

For a better approximation to ARC, the cache replacement policy should have been: If T_1^0 contains p or more pages, then remove a page from T_1^0, else remove a page from $T_1^1 \cup T_2$. However, this would require maintaining the list T_1^0, which seems to entail a much higher overhead on a hit. Hence, we eschew the precision, and

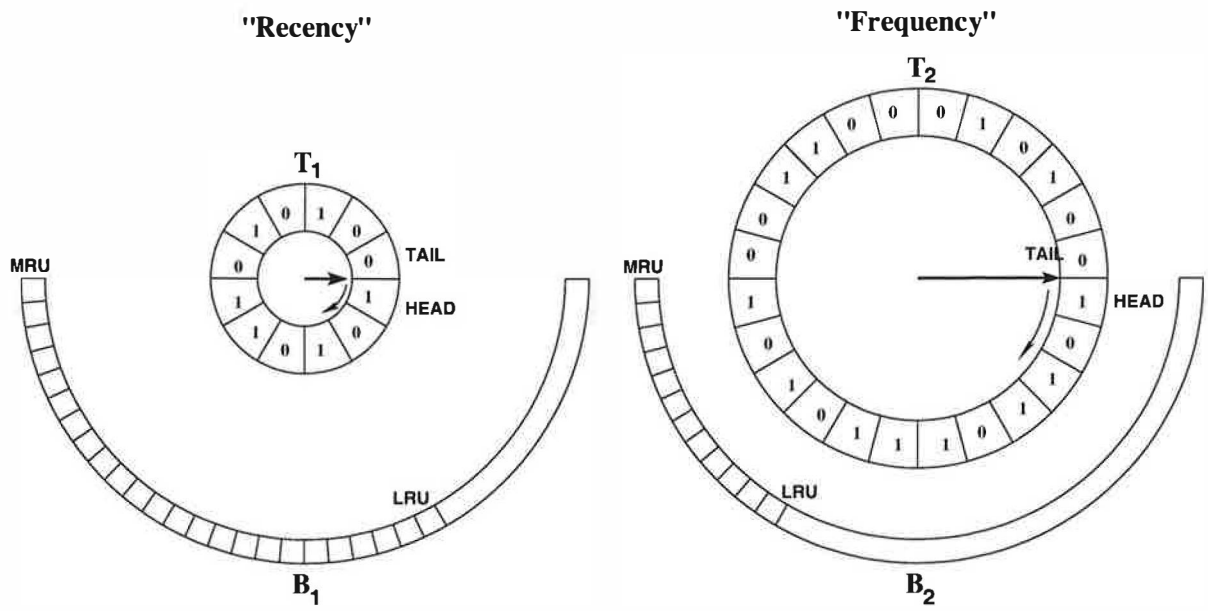


Fig. 1. A visual description of CAR. The CLOCKS T_1 and T_2 contain those pages that are in the cache and the lists B_1 and B_2 contain history pages that were recently evicted from the cache. The CLOCK T_1 captures "recency" while the CLOCK T_2 captures "frequency." The lists B_1 and B_2 are simple LRU lists. Pages evicted from T_1 are placed on B_1, and those evicted from T_2 are placed on B_2. The algorithm strives to keep B_1 to roughly the same size as T_2 and B_2 to roughly the same size as T_1. The algorithm also limits $|T_1| + |B_1|$ from exceeding the cache size. The sizes of the CLOCKs T_1 and T_2 are adapted continuously in response to a varying workload. Whenever a hit in B_1 is observed, the target size of T_1 is incremented; similarly, whenever a hit in B_2 is observed, the target size of T_1 is decremented. The new pages are inserted in either T_1 or T_2 immediately behind the clock hands which are shown to rotate clockwise. The page reference bit of new pages is set to 0. Upon a cache hit to any page in $T_1 \cup T_2$, the page reference bit associated with the page is simply set to 1. Whenever the T_1 clock hand encounters a page with a page reference bit of 1, the clock hand moves the page behind the T_2 clock hand and resets the page reference bit to 0. Whenever the T_1 clock hand encounters a page with a page reference bit of 0, the page is evicted and is placed at the MRU position in B_1. Whenever the T_2 clock hand encounters a page with a page reference bit of 1, the page reference bit is reset to 0. Whenever the T_2 clock hand encounters a page with a page reference bit of 0, the page is evicted and is placed at the MRU position in B_2.

go ahead with the above approximate policy where T_1' is used as an approximation to T_1^1.

The cache history replacement policy is simple as well:

> If $|T_1| + |B_1|$ contains exactly c pages, then remove a history page from B_1, else remove a history page from B_2.

Once again, for a better approximation to ARC, the cache history replacement policy should have been: If $|T_1^0| + |B_1|$ contains exactly c pages, then remove a history page from B_1, else remove a history page from B_2. However, this would require maintaining the size of T_1^0 which would require additional processing on a hit, defeating the very purpose of avoiding lock contention.

We now examine the algorithm in Figure 2 in detail.

Line 1 checks whether there is a hit, and if so, then line 2 simply sets the page reference bit to one. Observe that there is no MRU operation akin to LRU or ARC involved. Hence, cache hits are not serialized behind a lock and virtually no overhead is involved. The key insight is that the MRU operation is delayed until a replacement must be done (lines 29 and 36).

Line 3 checks for a cache miss, and if so, then line 4 checks if the cache is full, and if so, then line 5 carries out the cache replacement by deleting a page from either T_1 or T_2. We will dissect the cache replacement policy "replace()" in detail a little bit later.

If there is a cache miss (line 3), then lines 6-10 examine whether a cache history needs to be replaced. In particular, (line 6) if the requested page is totally new, that is, not in B_1 or B_2, and $|T_1| + |B_1| = c$ then (line 7) a page in B_1 is discarded, (line 8) else if the page is totally new and the cache history is completely full, then (line 9) a page in B_2 is discarded.

Finally, if there is a cache miss (line 3), then lines 12-20 carry out movements between the lists and also

carry out the adaptation of the target size for T_1. In particular, (line 12) if the requested page is totally new, then (line 13) insert it at the tail of T_1 and set its page reference bit to zero, (line 14) else if the requested page is in B_1, then (line 15) we increase the target size for the list T_1 and (line 16) insert the requested page at the tail of T_2 and set its page reference bit to zero, and, finally, (line 17) if the requested page is in B_2, then (line 18) we decrease the target size for the list T_1 and (line 19) insert the requested page at the tail of T_2 and set its page reference bit to zero.

Our adaptation rule is essentially the same as that in ARC. The role of the adaptation is to "invest" in the list that is most likely to give the highest hit per additional page invested.

We now examine the cache replacement policy (lines 22-39) in detail. The cache replacement policy can only replace a page with a page reference bit of zero. So, line 22 declares that no such suitable victim page to replace is yet found, and lines 23-39 keep looping until they find such a page.

If the size of the list T_1 is at least p and it is not empty (line 24), then the policy examines the head of T_1 as a replacement candidate. If the page reference bit of the page at the head is zero (line 25), then we have found the desired page (line 26), we now demote it from the cache and move it to the MRU position in B_1 (line 27). Else (line 28) if the page reference bit of the page at the head is one, then we reset the page reference bit to one and move the page to the tail of T_2 (line 29).

On the other hand, (line 31) if the size of the list T_1 is less than p, then the policy examines the page at the head of T_2 as a replacement candidate. If the page reference bit of the head page is zero (line 32), then we have found the desired page (line 33), and we now demote it from the cache and move it to the MRU position in B_1 (line 34). Else (line 35) if the page reference bit of the head page is one, then we reset the page reference bit to zero and move the page to the tail of T_2 (line 36).

Observe that while no MRU operation is needed during a hit, if a page has been accessed and its page reference bit is set to one, then during replacement such pages will be moved to the tail end of T_2 (lines 29 and 36). In other words, CAR approximates ARC by performing a delayed and approximate MRU operation during cache replacement.

While we have alluded to a multi-threaded environment to motivate CAR, for simplicity and brevity, our final algorithm is decidedly single-threaded. A true, real-life implementation of CAR will actually be based on a non-demand-paging framework that uses a free buffer pool of pre-determined size.

Observe that while cache hits are not serialized, like CLOCK, cache misses are still serialized behind a global lock to ensure correctness and consistency of the lists T_1, T_2, B_1, and B_2. This miss serialization can be somewhat mitigated by a free buffer pool.

Our discussion of CAR is now complete.

IV. CART

A limitation of ARC and CAR is that two consecutive hits are used as a test to promote a page from "recency" or "short-term utility" to "frequency" or "long-term utility". At upper level of memory hierarchy, we often observe two or more successive references to the same page fairly quickly. Such quick successive hits are known as "correlated references" [12] and are typically not a guarantee of long-term utility of a pages, and, hence, such pages can cause cache pollution–thus reducing performance. The motivation behind CART is to create a temporal filter that imposes a more stringent test for promotion from "short-term utility" to "long-term utility". The basic idea is to maintain a *temporal locality window* such that pages that are re-requested within the window are of short-term utility whereas pages that are re-requested outside the window are of long-term utility. Furthermore, the temporal locality window is itself an adaptable parameter of the algorithm.

The basic idea is to maintain four lists, namely, T_1, T_2, B_1, and B_2 as before. The pages in T_1 and T_2 are in the cache whereas the pages in B_1 and B_2 are only in the cache history. For simplicity, we will assume that T_1 and T_2 are implemented as Second Chance lists, but, in practice, they would be implemented as CLOCKs. The lists B_1 and B_2 are simple LRU lists. While we have used the same notation for the four lists, they will now be provided with a totally different meaning than that in either ARC or CAR.

Analogous to the invariants I1–I7 that were imposed on CAR, we now impose the same invariants on CART except that I2 and I3 are replaced, respectively, by:

I2' $\quad 0 \leq |T_2| + |B_2| \leq c.$
I3' $\quad 0 \leq |T_1| + |B_1| \leq 2c.$

As for CAR and CLOCK, for each page in $T_1 \cup T_2$ we will maintain a page reference bit. In addition, each page is marked with a *filter* bit to indicate whether it has *long-term utility* (say, "L") or only *short-term utility* (say, "S"). No operation on this bit will be required during a cache hit. We now detail manipulation and use of the filter bit. Denote by x a requested page.

1) Every page in T_2 and B_2 must be marked as "L".
2) Every page in B_1 must be marked as "S".
3) A page in T_1 could be marked as "S" or "L".
4) A head page in T_1 can only be replaced if its page reference bit is set to 0 and its filter bit is set to "S".

INITIALIZATION: Set $p = 0$ and set the lists T_1, B_1, T_2, and B_2 to empty.

CAR(x)
INPUT: The requested page x.

```
1:  if (x is in T1 ∪ T2) then /* cache hit */
2:      Set the page reference bit for x to one.
3:  else /* cache miss */
4:      if (|T1| + |T2| = c) then
            /* cache full, replace a page from cache */
5:          replace()
            /* cache directory replacement */
6:          if ((x is not in B1 ∪ B2) and (|T1| + |B1| = c)) then
7:              Discard the LRU page in B1.
8:          elseif ((|T1| + |T2| + |B1| + |B2| = 2c) and (x is not in B1 ∪ B2)) then
9:              Discard the LRU page in B2.
10:         endif
11:     endif
        /* cache directory miss */
12:     if (x is not in B1 ∪ B2) then
13:         Insert x at the tail of T1. Set the page reference bit of x to 0.
        /* cache directory hit */
14:     elseif (x is in B1) then
15:         Adapt: Increase the target size for the list T1 as: p = min{p + max{1, |B2|/|B1|}, c}
16:         Move x at the tail of T2. Set the page reference bit of x to 0.
        /* cache directory hit */
17:     else /* x must be in B2 */
18:         Adapt: Decrease the target size for the list T1 as: p = max{p − max{1, |B1|/|B2|}, 0}
19:         Move x at the tail of T2. Set the page reference bit of x to 0.
20:     endif
21: endif
```

```
replace()
22: found = 0
23: repeat
24:     if (|T1| >= max(1, p)) then
25:         if (the page reference bit of head page in T1 is 0) then
26:             found = 1;
27:             Demote the head page in T1 and make it the MRU page in B1.
28:         else
29:             Set the page reference bit of head page in T1 to 0, and make it the tail page in T2.
30:         endif
31:     else
32:         if (the page reference bit of head page in T2 is 0), then
33:             found = 1;
34:             Demote the head page in T2 and make it the MRU page in B2.
35:         else
36:             Set the page reference bit of head page in T2 to 0, and make it the tail page in T2.
37:         endif
38:     endif
39: until (found)
```

Fig. 2. Algorithm for Clock with Adaptive Replacement. This algorithm is self-contained. No tunable parameters are needed as input to the algorithm. We start from an empty cache and an empty cache directory. The **first key point** of the above algorithm is the simplicity of line 2, where cache hits are not serialized behind a lock and virtually no overhead is involved. The **second key point** is the continual adaptation of the target size of the list T_1 in lines 16 and 19. The **final key point** is that the algorithm requires no magic, tunable parameters as input.

5) If the head page in T_1 is of type "L", then it is moved to the tail position in T_2 and its page reference bit is set to zero.
6) If the head page in T_1 is of type "S" and has page reference bit set to 1, then it is moved to the tail position in T_1 and its page reference bit is set to zero.
7) A head page in T_2 can only be replaced if its page reference bit is set to 0.
8) If the head page in T_2 has page reference bit set to 1, then it is moved to the tail position in T_1 and its page reference bit is set to zero.
9) If $x \notin T_1 \cup B_1 \cup T_2 \cup B_2$, then set its type to "S."
10) If $x \in T_1$ and $|T_1| \geq |B_1|$, change its type to "L."
11) If $x \in T_2 \cup B_2$, then leave the type of x unchanged.
12) If $x \in B_1$, then x must be of type "S", change its type to "L."

When a page is removed from the cache directory, that is, from the set $T_1 \cup B_1 \cup T_2 \cup B_2$, its type is forgotten. In other words, a totally new page is put in T_1 and initially granted the status of "S", and this status is not upgraded upon successive hits to the page in T_1, but only upgraded to "L" if the page is eventually demoted from the cache and a cache hit is observed to the page while it is in the history list B_1. This rule ensures that there are two references to the page that are temporally separated by at least the length of the list T_1. Hence, the length of the list T_1 is the temporal locality window. The intent of the policy is to ensure that the $|T_1|$ pages in the list T_1 are the most recently used $|T_1|$ pages. Of course, this can only be done approximately given the limitation of CLOCK. Another source of approximation arises from the fact that a page in T_2, upon a hit, cannot immediately be moved to T_1.

While, at first sight, the algorithm appears very technical, the key insight is very simple: The list T_1 contains $|T_1|$ pages either of type "S" or "L", and is an approximate representation of "recency". The list T_2 contains remaining pages of type "L" that may have "long-term utility". In other words, T_2 attempts to capture useful pages which a simple recency based criterion may not capture.

We will adapt the temporal locality window, namely, the size of the list T_1, in a workload-dependent, adaptive, online fashion. Let p denote the target size for the list T_1. When p is set to the cache size c, the policy CART will coincide with the policy LRU.

The policy CART decides which list to delete from according to the rule in lines 36-40 of Figure 3. We also maintain a second parameter q which is the target size for the list B_1. The replacement rule for the cache history is described in lines 6-10 of Figure 3.

Let counters n_S and n_L denote the number of pages in the cache that have their filter bit set to "S" and "L", respectively. Clearly, $0 \leq n_S + n_L \leq c$, and, once the cache is full, $n_S + n_L = c$. The algorithm attempts to keep $n_S + |B_1|$ and $n_L + |B_2|$ to roughly c pages each.

The complete policy CART is described in Figure 3. We now examine the algorithm in detail.

Line 1 checks for a hit, and if so, line 2 simply sets the page reference bit to one. This operation is exactly similar to that of CLOCK and CAR and gets rid of the the need to perform MRU processing on a hit.

Line 3 checks for a cache miss, and if so, then line 4 checks if the cache is full, and if so, then line 5 carries out the cache replacement by deleting a page from either T_1 or T_2. We dissect the cache replacement policy "replace()" in detail later.

If there is a cache miss (line 3), then lines 6-10 examine whether a cache history needs to be replaced. In particular, (line 6) if the requested page is totally new, that is, not in B_1 or B_2, $|B_1| + |B_2| = c + 1$, and B_1 exceeds its target, then (line 7) a page in B_1 is discarded, (line 8) else if the page is totally new and the cache history is completely full, then (line 9) a page in B_2 is discarded.

Finally, if there is a cache miss (line 3), then lines 12-21 carry out movements between the lists and also carry out the adaptation of the target size for T_1. In particular, (line 12) if the requested page is totally new, then (line 13) insert it at the tail of T_1, set its page reference bit to zero, set the filter bit to "S", and increment the counter n_S by 1. (Line 14) Else if the requested page is in B_1, then (line 15) we increase the target size for the list T_1 (increase the temporal window) and insert the requested page at the tail end of T_1 and (line 16) set its page reference bit to zero, and, more importantly, also changes its filter bit to "L". Finally, (line 17) if the requested page is in B_2, then (line 18) we decrease the target size for the list T_1 and insert the requested page at the tail end of T_1, (line 19) set its page reference bit to zero, and (line 20) update the target q for the list B_1.

The essence of the adaptation rule is: On a hit in B_1, it favors increasing the size of T_1, and, on a hit in B_2, it favors decreasing the size of T_1.

Now, we describe the "replace()" procedure. (Lines 23-26) While the page reference bit of the head page in T_2 is 1, then move the page to the tail position in T_1, and also update the target q to control the size of B_1. In other words, these lines capture the movement from T_2 to T_1. When this while loop terminates, either T_2 is empty, or the page reference bit of the head page in T_2 is set to 0, and, hence, can be removed from the cache if desired.

(Line 27-35) While the filter bit of the head page in T_1 is "L" or the page reference bit of the head page in T_1 is 1, keep moving these pages. When this while loop

terminates, either T_1 will be empty, or the head page in T_1 has its filter bit set to "S" and page reference bit set to 0, and, hence, can be removed from the cache if desired. (Lines 28-30) If the page reference bit of the head page in T_1 is 1, then make it the tail page in T_1. At the same time, if B_1 is very small or T_1 is larger than its target, then relax the temporal filtering constraint and set the filter bit to "L". (Lines 31-33) If the page reference bit is set to 0 but the filter bit is set to "L", then move the page to the tail position in T_2. Also, change the target B_1.

(Lines 36-40) These lines represent our cache replacement policy. If T_1 contains at least p pages and is not empty, then remove the head page in T_1, else remove the head page in T_2.

Our discussion of CART is now complete.

V. Experimental Results

In this section, we will focus our experimental simulations to compare LRU, CLOCK, ARC, CAR, and CART.

A. Traces

Table I summarizes various traces that we used in this paper. These traces are the same as those in [10, Section V.A], and, for brevity, we refer the reader there for their description. These traces capture disk accesses by databases, web servers, NT workstations, and a synthetic benchmark for storage controllers. All traces have been filtered by up-stream caches, and, hence, are representative of workloads seen by storage controllers, disks, or RAID controllers.

Trace Name	Number of Requests	Unique Pages
P1	32055473	2311485
P2	12729495	913347
P3	3912296	762543
P4	19776090	5146832
P5	22937097	3403835
P6	12672123	773770
P7	14521148	1619941
P8	42243785	977545
P9	10533489	1369543
P10	33400528	5679543
P11	141528425	4579339
P12	13208930	3153310
P13	15629738	2497353
P14	114990968	13814927
ConCat	490139585	47003313
Merge(P)	490139585	47003313
DS1	43704979	10516352
SPC1	41351279	6050363
S1	3995316	1309698
S2	17253074	1693344
S3	16407702	1689882
Merge (S)	37656092	4692924

TABLE I. A summary of various traces used in this paper. Number of unique pages in a trace is termed its "footprint".

For all traces, we only considered the read requests. All hit ratios reported in this paper are *cold start*. We will report hit ratios in percentages (%).

B. Results

In Table II, we compare LRU, CLOCK, ARC, CAR, and CART for the traces SPC1 and Merge(S) for various cache sizes. It can be clearly seen that CLOCK has performance very similar to LRU, and CAR/CART have performance very similar to ARC. Furthermore, CAR/CART substantially outperform CLOCK.

SPC1

c (pages)	LRU	CLOCK	ARC	CAR	CART
65536	0.37	0.37	0.82	0.84	0.90
131072	0.78	0.77	1.62	1.66	1.78
262144	1.63	1.63	3.23	3.29	3.56
524288	3.66	3.64	7.56	7.62	8.52
1048576	9.19	9.31	20.00	20.00	21.90

Merge(S)

c (pages)	LRU	CLOCK	ARC	CAR	CART
16384	0.20	0.20	1.04	1.03	1.10
32768	0.40	0.40	2.08	2.07	2.20
65536	0.79	0.79	4.07	4.05	4.27
131072	1.59	1.58	7.78	7.76	8.20
262144	3.23	3.27	14.30	14.25	15.07
524288	8.06	8.66	24.34	24.47	26.12
1048576	27.62	29.04	40.44	41.00	41.83
1572864	50.86	52.24	57.19	57.92	57.64
2097152	68.68	69.50	71.41	71.71	71.77
4194304	87.30	87.26	87.26	87.26	87.26

TABLE II. A comparison of hit ratios of LRU, CLOCK, ARC, CAR, and CART on the traces SPC1 and Merge(S). All hit ratios are reported in percentages. The page size is 4 KBytes for both traces. The largest cache simulated for SPC1 was 4 GBytes and that for Merge(S) was 16 GBytes. It can be seen that LRU and CLOCK have similar performance, while ARC, CAR, and CART also have similar performance. It can be seen that ARC/CAR/CART outperform LRU/CLOCK.

In Figures 4 and 5, we graphically compare the hit-ratios of CAR to CLOCK for all of our traces. The performance of CAR was very close to ARC and CART and the performance of CLOCK was very similar to LRU, and, hence, to avoid clutter, LRU, ARC, and CART are not plotted. It can be clearly seen that across a wide variety of workloads and cache sizes CAR outperforms CLOCK–sometimes quite dramatically.

Finally, in Table III, we produce an at-a-glance-summary of LRU, CLOCK, ARC, CAR, and CART for various traces and cache sizes. Once again, the same conclusions as above are seen to hold: ARC, CAR, and CART outperform LRU and CLOCK, ARC, CAR, and CART have a very similar performance, and CLOCK has performance very similar to LRU.

INITIALIZATION: Set $p = 0$, $q = 0$, $n_S = n_L = 0$, and set the lists T_1, B_1, T_2, and B_2 to empty.

CART(x)
INPUT: The requested page x.
1: if (x is in $T_1 \cup T_2$) then /* cache hit */
2: Set the page reference bit for x to one.
3: else /* cache miss */
4: if ($|T_1| + |T_2| = c$) then
/* cache full, replace a page from cache */
5: replace()
/* history replacement */
6: if (($x \notin B_1 \cup B_2$) and ($|B_1| + |B_2| = c + 1$) and (($|B_1| > \max\{0, q\}$) or (B_2 is empty))) then
7: Remove the bottom page in B_1 from the history.
8: elseif (($x \notin B_1 \cup B_2$) and ($|B_1| + |B_2| = c + 1$)) then
9: Remove the bottom page in B_2 from the history.
10: endif
11: endif
/* history miss */
12: if (x is not in $B_1 \cup B_2$) then
13: Insert x at the tail of T_1. Set the page reference bit of x to 0, set filter bit of x to "S", and $n_S = n_S + 1$.
/* history hit */
14: elseif (x is in B_1) then
15: Adapt: Increase the target size for the list T_1 as: $p = \min\{p + \max\{1, n_S/|B_1|\}, c\}$. Move x to the tail of T_1.
16: Set the page reference bit of x to 0. Set $n_L = n_L + 1$. Set type of x to "L".
/* history hit */
17: else /* x must be in B_2 */
18: Adapt: Decrease the target size for the list T_1 as: $p = \max\{p - \max\{1, n_L/|B_2|\}, 0\}$. Move x to the tail of T_1.
19: Set the page reference bit of x to 0. Set $n_L = n_L + 1$.
20: if ($|T_2| + |B_2| + |T_1| - n_S \geq c$) then, Set target $q = \min(q + 1, 2c - |T_1|)$, endif
21: endif
22: endif

replace()
23: while (the page reference bit of the head page in T_2 is 1)) then
24: Move the head page in T_2 to tail position in T_1. Set the page reference bit to 0.
25: if ($|T_2| + |B_2| + |T_1| - n_S \geq c$) then, Set target $q = \min(q + 1, 2c - |T_1|)$, endif
26: endwhile
/* The following while loop should stop, if T_1 is empty */
27: while ((the filter bit of the head page in T_1 is "L") or (the page reference bit of the head page in T_1 is 1))
28: if ((the page reference bit of the head page in T_1 is 1)
29: Move the head page in T_1 to tail position in T_1. Set the page reference bit to 0.
30: if (($|T_1| \geq \min(p + 1, |B_1|)$) and (the filter bit of the moved page is "S")) then,
set type of x to "L", $n_S = n_S - 1$, and $n_L = n_L + 1$.
endif
31: else
32: Move the head page in T_1 to tail position in T_2. Set the page reference bit to 0.
33: Set $q = \max(q - 1, c - |T_1|)$.
34: endif
35: endwhile
36: if ($|T_1| >= \max(1, p)$) then
37: Demote the head page in T_1 and make it the MRU page in B_1. $n_S = n_S - 1$.
38: else
39: Demote the head page in T_2 and make it the MRU page in B_2. $n_L = n_L - 1$.
40: endif

Fig. 3. Algorithm for Clock with Adaptive Replacement and Temporal Filtering. This algorithm is self-contained. No tunable parameters are needed as input to the algorithm. We start from an empty cache and an empty cache history.

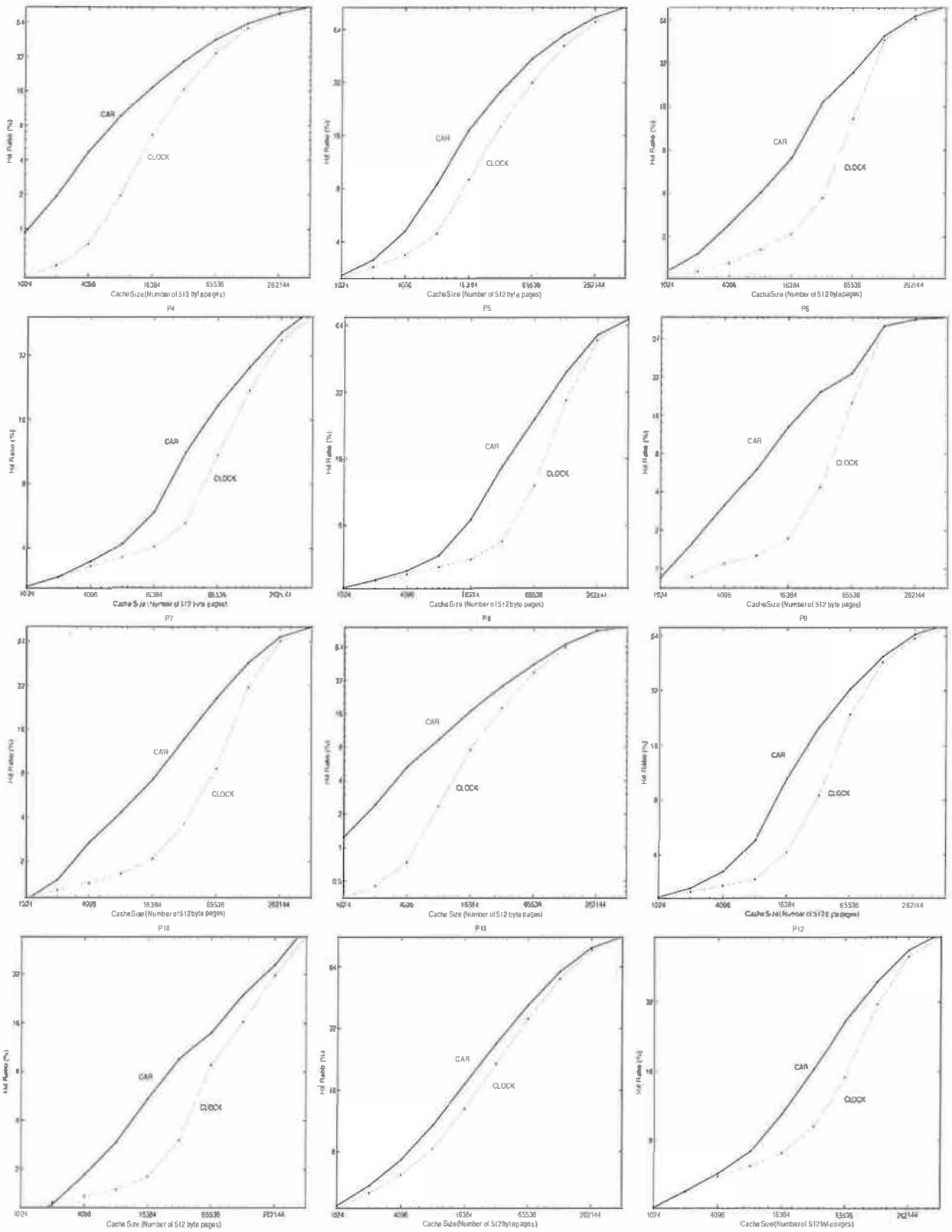


Fig. 4. A plot of hit ratios (in percentages) achieved by CAR and CLOCK. Both the x- and y-axes use logarithmic scale.

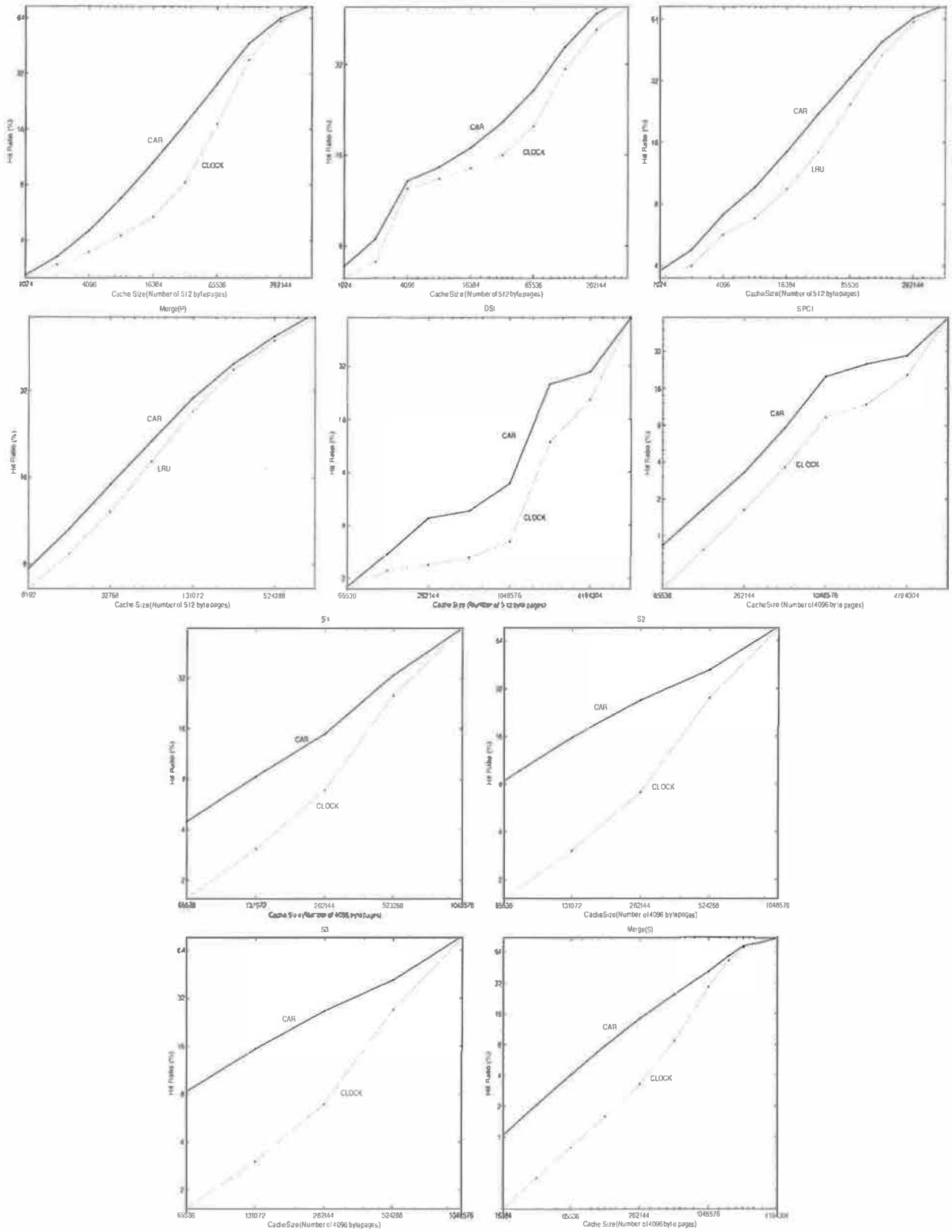


Fig. 5. A plot of hit ratios (in percentages) achieved by CAR and CLOCK. Both the x- and y-axes use logarithmic scale.

Workload	c (pages)	space (MB)	LRU	CLOCK	ARC	CAR	CART
P1	32768	16	16.55	17.34	28.26	29.17	29.83
P2	32768	16	18.47	17.91	27.38	28.38	28.63
P3	32768	16	3.57	3.74	17.12	17.21	17.54
P4	32768	16	5.24	5.25	11.24	11.22	9.25
P5	32768	16	6.73	6.78	14.27	14.78	14.77
P6	32768	16	4.24	4.36	23.84	24.34	24.53
P7	32768	16	3.45	3.62	13.77	13.86	14.79
P8	32768	16	17.18	17.99	27.51	28.21	28.97
P9	32768	16	8.28	8.48	19.73	20.09	20.75
P10	32768	16	2.48	3.02	9.46	9.63	9.71
P11	32768	16	20.92	21.51	26.48	26.99	27.26
P12	32768	16	8.93	9.18	15.94	16.25	16.41
P13	32768	16	7.83	8.26	16.60	17.09	17.74
P14	32768	16	15.73	15.98	20.52	20.59	20.63
ConCat	32768	16	14.38	14.79	21.67	22.06	22.24
Merge(P)	262144	128	38.05	38.60	39.91	39.90	40.12
DS1	2097152	1024	11.65	11.86	22.52	25.31	21.12
SPC1	1048576	4096	9.19	9.31	20.00	20.00	21.91
S1	524288	2048	23.71	25.26	33.43	33.42	33.62
S2	524288	2048	25.91	27.84	40.68	41.86	42.10
S3	524288	2048	25.26	27.13	40.44	41.67	41.87
Merge(S)	1048576	4096	27.62	29.04	40.44	41.01	41.83

TABLE III. At-a-glance comparison of hit ratios of LRU, CLOCK, ARC, CAR, and CART for various workloads. All hit ratios are reported in percentages. It can be seen that LRU and CLOCK have similar performance, while ARC, CAR, and CART also have similar performance. It can be seen that ARC, CAR, and CART outperform LRU and CLOCK–sometimes quite dramatically.

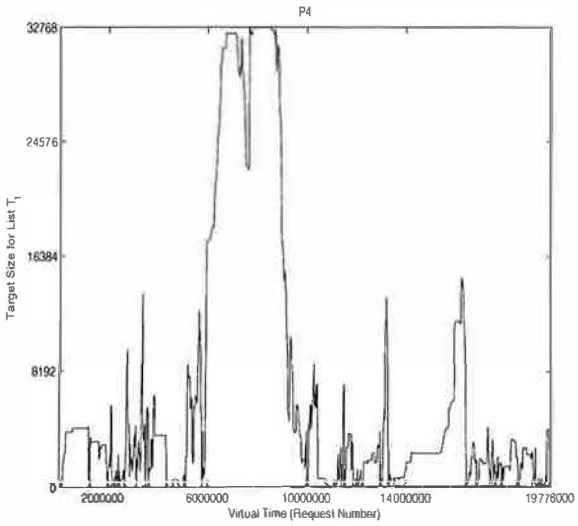


Fig. 6. A plot of the adaptation parameter p (the target size for list T_1) versus the virtual time for the algorithm CAR. The trace is P4, the cache size is 32768 pages, and the page size is 512 bytes.

VI. Conclusions

In this paper, by combining ideas and best features from CLOCK and ARC we have introduced a policy CAR that removes disadvantages D1, D2, D3 and D4 of LRU.

CAR **removes the cache hit serialization problem of** LRU **and** ARC.

CAR **has a very low overhead on cache hits**.

CAR **is self-tuning**. The policy CAR requires no tunable, magic parameters. It has one tunable parameter p that balances between recency and frequency. The policy adaptively tunes this parameter–in response to an evolving workload–so as to increase the hit-ratio. A closer examination of the parameter p shows that it can fluctuate from recency ($p = c$) to frequency ($p = 0$) and back– all within a single workload. In other words, adaptation really matters! Also, it can be shown that CAR performs as well as its offline counterpart which is allowed to select the best, offline, fixed value of p chosen specifically for a given workload and a cache size. In other words, adaptation really works! See Figure 6 for a graphical demonstration of how p fluctuates. The self-tuning nature of CAR makes it very attractive for deployment in environments where no *a priori* knowledge of the workloads is available.

CAR **is scan-resistant**. A scan is any sequence of one-time use requests. Such requests will be put on top of the list T_1 and will eventually exit from the cache without polluting the high-quality pages in T_2. Moreover, in presence of scans, there will be relatively fewer hits in B_1 as compared to B_2. Hence, our adaptation rule will tend to further increase the size of T_2 at the expense of T_1, thus further decreasing the residency time of scan in even T_1.

CAR **is high-performance**. CAR outperforms LRU and CLOCK on a wide variety of traces and cache sizes, and has performance very comparable to ARC.

CAR has low space overhead, typically, less that 1%.

CAR is simple to implement. Please see Figure 2.

CART has temporal filtering. The algorithm CART has all the above advantages of CAR, but, in addition, it employs a much stricter and more precise criterion to distinguish pages with short-term utility from those with long-term utility.

It should be clear from Section II-A that a large number of attempts have been made to improve upon LRU. In contrast, relatively few attempts have been made to improve upon CLOCK–the most recent being in 1978! We believe that this is due to severe constraints imposed by CLOCK on how much processing can be done on a hit and its removal of the single global lock. Genuine new insights were required to invent novel, effective algorithms that improve upon CLOCK. We hope that CAR and CART represents two such fundamental insights and that they will be seriously considered by cache designers.

ACKNOWLEDGMENT

We are grateful to Bruce Lindsay and Honesty Young for suggesting that we look at lock contention, to Frank Schmuck for pointing out bugs in our previous attempts, and to Pawan Goyal for urging us to publish this work. We are grateful to our manager, Moidin Mohiuddin, for his constant support and encouragement during this work. The second author is grateful to Nimrod Megiddo for his collaboration on ARC. We are grateful to Bruce McNutt and Renu Tewari for the SPC1 trace, to Windsor Hsu for traces P1 through P14, to Ruth Azevedo for the trace DS1, and to Ken Bates and Bruce McNutt for traces S1-S3. We are indebted to Binny Gill for drawing the beautiful and precise Figure 1.

REFERENCES

[1] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Sys. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[2] M. J. Bach, *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

[3] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*. Prentice-Hall, 1997.

[4] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Reading, MA: Addison-Wesley, 1995.

[5] J. E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[6] F. J. Corbató, "A paging experiment with the multics system," in *In Honor of P. M. Morse*, pp. 217–228, MIT Press, 1969. Also as MIT Project MAC Report MAC-M-384, May 1968.

[7] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita, "Starburst mid-flight: As the dust clears," *IEEE Trans. Knowledge and Data Engineering*, vol. 2, no. 1, pp. 143–160, 1990.

[8] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[9] H. Levy and P. H. Lipman, "Virtual memory management in the VAX/VMS operating system," *IEEE Computer*, pp. 35–41, March 1982.

[10] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST 03), San Franciso, CA*, pp. 115–130, 2003.

[11] N. Megiddo and D. S. Modha, "One up on LRU," *;login – The Magazine of the USENIX Association*, vol. 28, pp. 7–11, August 2003.

[12] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proc. ACM SIGMETRICS Conf.*, pp. 134–142, 1990.

[13] A. J. Smith, "Bibliography on paging and related topics," *Operating Systems Review*, vol. 12, pp. 39–56, 1978.

[14] A. J. Smith, "Second bibliography for cache memories," *Computer Architecture News*, vol. 19, no. 4, 1991.

[15] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," *J. ACM*, vol. 18, no. 1, pp. 80–93, 1971.

[16] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *Proc. ACM SIGMOD Conf.*, pp. 297–306, 1993.

[17] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. VLDB Conf.*, pp. 297–306, 1994.

[18] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies," in *Proc. ACM SIGMETRICS Conf.*, pp. 134–143, 1999.

[19] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Trans. Computers*, vol. 50, no. 12, pp. 1352–1360, 2001.

[20] Y. Zhou and J. F. Philbin, "The multi-queue replacement algorithm for second level buffer caches," in *Proc. USENIX Annual Tech. Conf. (USENIX 2001), Boston, MA*, pp. 91–104, June 2001.

[21] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proc. ACM SIGMETRICS Conf.*, 2002.

[22] W. R. Carr and J. L. Hennessy, "WSClock – a simple and effective algorithm for virtual memory management," in *Proc. Eighth Symp. Operating System Principles*, pp. 87–95, 1981.

[23] H. T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *Proceedings of the 11th International Conference on Very Large Databases, Stockholm, Sweden*, pp. 127–141, 1985.

[24] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Trans. Database Systems*, vol. 3, no. 3, pp. 223–247, 1978.

[25] V. F. Nicola, A. Dan, and D. M. Dias, "Analysis of the generalized clock buffer replacement scheme for database transaction processing," in *ACM SIGMETRICS*, pp. 35–46, 1992.

[26] U. Vahalia, *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.

Circus: Opportunistic Block Reordering for Scalable Content Servers

Stergios V. Anastasiadis Rajiv G. Wickremesinghe Jeffrey S. Chase
Department of Computer Science, Duke University
Durham, NC 27708, USA
{stergios,rajiv,chase}@cs.duke.edu

Abstract

Whole-file transfer is a basic primitive for Internet content dissemination. Content servers are increasingly limited by disk arm movement given the rapid growth in disk density, disk transfer rates, server network bandwidth, and content size. Individual file transfers are sequential, but the block access sequence on a content server is effectively random when many slow clients access large files concurrently. Although larger blocks can help improve disk throughput, buffering requirements increase linearly with block size.

This paper explores a novel block reordering technique that can reduce server disk traffic significantly when large content files are shared. The idea is to transfer blocks to each client in any order that is convenient for the server. The server sends blocks to each client opportunistically in order to maximize the advantage from the disk reads it issues to serve other clients accessing the same file. We first illustrate the motivation and potential impact of opportunistic block reordering using a simple analytical model. Then we describe a file transfer system using a simple block reordering algorithm, called Circus. Experimental results with the Circus prototype show that it can improve server throughput by a factor of two or more in workloads with strong file access locality.

1 Introduction

Many Internet services are based on whole-file transfers or *file downloads*. For our purposes, the file download primitive has three defining characteristics: (i) each client initiates transfer of an entire file, rather than a block or fragment, (ii) the client postpones interpretation of the data until the transfer completes, and (iii) the transfer may occur as rapidly as permitted by the server, the client, and the network resources. Whole-file transfer is a building block of peer-to-peer(P2P) file sharing, the Web, media services, and data grids [2].

Conventional file download protocols such as FTP or HTTP transfer each file as an ordered stream of bytes, and use the underlying transport protocol (e.g., TCP) to deliver data in sequence. Nevertheless, the semantics of file download permit the server to deliver data to each client in any order, as long as the client eventually receives the entire file. Reordering relies on the client to reassemble the content, similar to P2P "content swarming" systems that disperse file blocks across multiple servers (such as BitTorrent [14]).

This paper investigates *opportunistic* block reordering as a technique to improve the cache effectiveness of a content server. More effective caching can increase the server throughput for a given disk subsystem, and decrease the disk and memory cost needed to fill the server network link. Block reordering complements well-known techniques to improve cache effectiveness with improved replacement algorithms (e.g., [20]) or integrated caching and prefetching [10]. Most obviously, it changes the block access sequence to suit the needs of the storage system, instead of just managing the storage cache to suit a specific block access sequence. In essence, block reordering extends the server memory by using the client memory as a reassembly buffer.

Block reordering is helpful primarily when multiple clients request large shared files concurrently. Workload studies have shown this case to be quite common and important for overall performance. Typically a modest percentage of popular files receive most of the requests in a content server, and larger files account for a disproportionally large share of the data traffic [3, 5, 7, 11, 27]. A recent workload characterization of a popular P2P system concluded that 42% of the data requested from a typical academic site involved transfers of a few hundred large objects with an average size of several hundred megabytes each [27]. In that study, large object caching on its own could yield a byte hit ratio as high as 38%. Block reordering is especially promising in content networks that employ content-based request routing to concentrate the requests for each file on a small set of servers; recent studies show large improvements in server cache locality from this technique [16, 22, 30].

When files are small it is sufficient to cache them in their entirety to capture most of the benefit from sharing. But caching of large files is less effective because they consume more cache space and therefore they are more vulnerable to eviction before the next request can generate a cache hit. As file size increases relative to the server memory, the hit ratios degrade and the disk access transaction rate limits server throughput [6, 27]. Unfortunately, ongoing advances in disk bandwidth—due to in-

creasing areal density and faster rotation speeds—do not help appreciably. In fact, seek overheads dominate because concurrent requests from a large number of clients tend to destroy the inherent sequentiality of block accesses to each file, producing a block access sequence that is effectively random. Since seek times improve more slowly than disk bandwidth, faster disks actually make the problem worse because these seek overheads consume a larger share of the arm time.

Several other techniques are directed at serving large shared content files, primarily for continuous media. One solution is to abandon caching and use large disk transfers to reduce seeking, a technique that is well-studied for streaming video servers (e.g., [4]). But buffer demands increase linearly with transfer size and the number of clients. A second alternative is to encode the shared files using forward error correcting codes (FEC) [9, 26], e.g., the Tornado codes used by Digital Fountain. FEC yields optimal caching effectiveness, since every receiver can benefit from each block fetched from the disk. Unfortunately, FEC increases the volume of data needed to store and transmit a stream by as much as an order of magnitude, depending on the skew of client transfer rates. Other techniques include stream merging methods [17, 18]. Section 6 discusses related work in more detail.

The rest of this paper is organized as follows. Section 2 uses a simple model to explore the performance behavior of file download servers and motivate opportunistic block reordering. Section 3 proposes a simple parameterized block selection algorithm for reordering. Section 4 describes its implementation in the Circus content server, and Section 5 presents experimental results. Section 6 sets opportunistic block reordering in context with previous work, and Section 7 summarizes our conclusions.

2 Overview and Motivation

We first define the *file download problem* more formally, and then outline the goals and motivation for our work. Consider a content server with a network link of bandwidth R_n bytes/s and a population of N clients concurrently downloading shared files. We use the term *file* to mean a unique ordered set of data blocks; thus it could refer to any segment of a larger data set. Suppose that each client i can receive data at a sustained rate $R_c^i \leq R_n$, with an average client rate R_c bytes/s. The goal is to schedule block reads from disks and block transfers to clients in order to maximize $X(N)$, the system throughput for N concurrent requests, or the number of download requests completed per time unit. Unless it is saturated, the server should be fair and it should complete each request in the minimum time that the network allows: a request from client i to download a file of length L bytes should complete after approximate delay L/R_c^i.

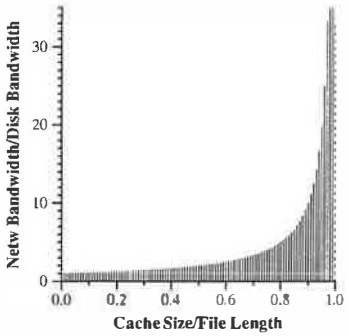


Figure 1: The maximum ratio of network bandwidth to disk bandwidth, as a function of the portion of the data set size that fits in the cache. The system is heavily disk-limited unless the bulk of the data set fits in memory.

Since this paper focuses on the design of the server and its storage system, we will claim success when the server is network-limited, i.e., it consumes all of its available network bandwidth. Suppose the average file length is L bytes. For low client loads, the server may be limited by the aggregate network bandwidth to its clients: $X(N) \leq R_c N/L$. If the client population is large, or if the clients have high-speed links, then the server may be limited by its own network link: $X(N) \leq R_n/L$ (with unicast). We assume without loss of generality that the server's CPU and memory system are able to sustain the peak data rate R_n when serving large files.

Our objective, then, is to minimize the memory and storage resources needed to feed content to the network at the peak rate R_n for a sufficiently large client population: $N \geq R_n/R_c$. Suppose the content server has a memory of size M and D disks delivering a peak bandwidth of R_d byte/s each. Our purpose is to explore how block reordering can reduce the number of disks D needed for a given M, and/or reduce the M needed for a given D and R_d. Equivalently, we may view the objective as maximizing the network speed R_n or client population size N that a given configuration (M, D) can support. The problem is uninteresting for small R_n, e.g., in serverless P2P systems that distribute server functions across many slow clients. The problem is most interesting for servers in data centers with high-capacity network links, serving large client populations with slow transfer rates R_c. N may be quite large for commercial content servers with a large bandwidth disparity relative to their clients: IP network bandwidth prices are dropping [13], while broadband deployments for the "last mile" to clients continue to lag behind. Although our approach does not require multicast support, it is compatible with multicast and N may be even larger when it exists.

2.1 Caching

The server's memory of size M consists of a common pool of shared buffers. A server uses these buffers for some combination of disk buffering, network buffering, and data caching. For example, suppose the server fetches data from storage in blocks of size B bytes. A typical server would buffer each fetched block until all client transmits of the block to clients have completed; the surplus memory acts as a cache over blocks recently fetched for some client i that are deemed likely to be needed soon for another client j requesting the same file. Caching is effective for small files that can reside in the cache in their entirety until the next request [6]. However, if j's request arrives t time units after i's, then the server has already fetched up to tR_c^i bytes of the file into memory and delivered them to i. The memory needed to cache the segment until j is ready to receive it grows with the inter-arrival time t. If $R_c^i > R_c^j$ then the required cache space continues to grow as the transfers progress—when the system is constrained to deliver the data in order to both clients, as for conventional file servers.

Since each file request accesses each block of the file exactly once, block accesses are uniformly distributed over the entire file length. As the number of clients increases, and as client rates and arrival times vary, the block access request stream shows less temporal locality and becomes effectively random. Of course, there is spatial locality when the server delivers each block in sequence; the system may exploit this by using a larger block size B, as discussed below (or, equivalently, by prefetching more deeply). Suppose without loss of generality that the content consists of a single file of length $L >> M$, or any number of equally popular files with aggregate size L. Then the probability of any block access being a cache hit is $P_{hit} = M/L$, and $P_{miss} = 1 - M/L$ is the probability that the access requires a disk fetch. Then the server's storage system must be capable of sustaining block fetches at rate $(R_n/B)(1 - M/L)$. Figure 1 shows the role of cache size in determining the number of disks required to sustain this rate.

2.2 Storage Throughput

The areal density of magnetic storage has been doubling every year, and disks today spin several times faster than ten years ago [13]. While these trends increase sequential disk bandwidth, throughput for random block accesses (IOPS) has not kept pace. Seek costs tend to dominate random accesses, and these costs are limited by mechanical movement and are not improving as rapidly. Unfortunately, the block miss stream for a content server degrades to random access as the client population N increases, for the same reasons outlined above.

One solution is to increase the block size B. This reduces the rate of disk operations required to sustain a given effective bandwidth, which is inversely proportional to B. This technique can significantly reduce the number of disks required. Suppose each disk has an average head positioning time per access of T_{pos} seconds (seek plus half-rotation). Every disk access moves a block of size B bytes, and takes time $T_{pos} + B/R_d$. Thus, each disk supports $\frac{1}{T_{pos}+B/R_d}$ random accesses per time unit, and the aggregate peak disk bandwidth for D disks becomes $X_d = \frac{B \times D}{T_{pos}+B/R_d}$ bytes/s. The server needs $D = \frac{R_n}{B/(T_{pos}+B/R_d)}$ disks to fully pipeline the network. For example, Figure 2 illustrates the disk random access throughput when $T_{pos} = 0.005$ s, a typical value. With block size $B = 256$ KB, and disks of $R_d = 50$ MB/s, the disk throughput X_d becomes roughly 25 MB/s. We need about $D = 5$ such disks to feed a server network link of $R_n = 1$ Gb/s.

However, Figure 3 shows that this approach consumes large amounts of memory with large client populations. Depending on the buffering scheme the minimum memory size M for N active clients is $NB/2 \leq M \leq 2NB$. The 1 Gb/s server can support $N = 664$ T1 (1.544 Mbps = 193 KB/s) client connections, consuming a minimum $M = 87$ MB just for device buffering with $B = 256$ KB. If we increase the block size to $B = 1$ MB then disk throughput becomes $X_d = 40$ MB/s, the number of disks drops to $D = 4$, and the minimum memory grows to $M = 325$ MB. If instead, we assume slower clients with $R_c = 56$ Kbps, the server needs a minimum $M = 2.18$ GB and $M = 8.7$ GB for block size $B = 256$ KB and $B = 1$ MB, respectively. In media servers using large block sizes it is common to eliminate the cache and partition memory into separate buffer regions for each client; each region is sufficient to hold any blocks in transit between the disk and the network for that client [4].

2.3 Summary

In this section we explained why accesses at the block level appear increasingly random as the client population N grows. This minimizes the benefit of conventional caching and weakens the locality of the disk accesses from the cache miss stream. The combination of these effects increases server cost, since more disks and/or more memory are needed to fill any given network link. For example, the experiments in this paper use disks with average sequential throughput of $R_d = 33$ MByte/s, but a conventional FTP server under even modest load delivers a per-disk throughput of roughly 10 MB/s, including cache hits. Thus we need about a dozen such disks to feed a network link of 1 Gb/s.

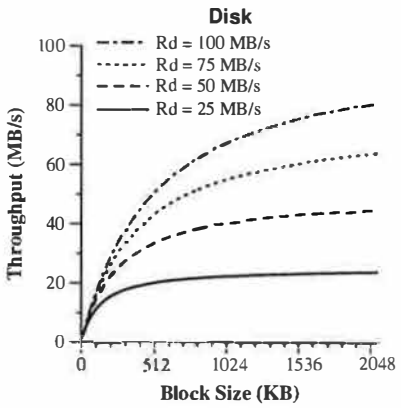


Figure 2: Disk throughput with random block accesses in a disk with positioning time $T_{pos} = 0.005$s across different transfer block sizes, and sequential disk transfer rates R_d. We show that as R_d increases, block size must exceed one megabyte to achieve peak disk throughput.

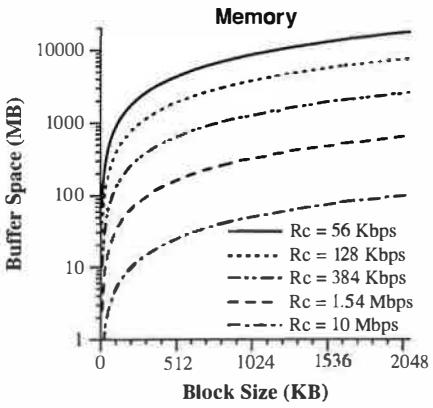


Figure 3: Minimum buffer space required to support clients at different rates in a server with a 1 Gbps network link. The minimum buffer space can reach several gigabytes for large populations of clients with bandwidth less than 1 Mbps.

3 Opportunistic Block Reordering

To support large client populations inexpensively, we need to minimize the number of disk fetches and maximize the number of clients served with each fetch. Ideally, when multiple clients request the same file, the server retrieves each block from disk only once and sends it to every client. Then, using the notation introduced previously, serving a file to N clients at network throughput NR_c requires disk throughput R_c and buffer space $B/2 \leq M \leq 2B$, all of which are independent of N.

A file download server must transfer all the blocks of a file to satisfy each file request; thus an arriving file request from client i reveals information about block accesses on behalf of client i for at least the next L/R_c^i time units. The system can use these revealed block sequences to improve performance and/or reduce cost. In particular, it can improve cache performance by opportunistically reordering the accesses, e.g., to send each fetched block opportunistically to all clients known to need the block in the future, before replacing it from memory. The server can send blocks out-of-order by attaching an application-layer header [12] to each transmitted block that specifies its offset; clients resequence the data according to these block headers. This section presents the block reordering algorithm in the *Circus* content server.

The *Circus* server maintains a list of files with downloads currently in progress (*active files*), and for each active file a list of clients currently downloading that file (*active clients*). For each active client the server creates a FIFO queue of references to the file blocks selected to send next to that client. The server transmits blocks from the head of the queue when the client's network transport send window opens. When a queue drains below a threshold, the server selects another block to transmit to that client, schedules a disk fetch if necessary, and adds the block reference to the queue tail. *Circus* strives to guarantee forward progress and fairness by serving all active clients at their maximum network transfer rates.

The block selection policy is the key to the *Circus* algorithm. For each active client, *Circus* also maintains a set data structure (bitmap) to record the set of blocks already sent to the client. Thus, for each block of an active file, it knows the set of active clients that have yet to receive the block. The selection policy could select blocks to greedily maximize the number of clients that benefit from each block, or minimize the number of block fetches needed to fill up all block queues at any given time. However, these policies are computationally complex, and their cost scales with the file size.

We propose a simple block selection policy that is fast and has modest memory requirements. Figure 5 outlines the algorithm. Its complexity scales linearly with the number of active clients for each file. To preserve locality of reference among the active clients, *Circus* designates an (*active region*) for each file as preferred for block selection and caching. The active region is a moving contiguous region within a circular block space. The *file cursor* specifies the front edge of the active region of a file. The length of a file's active region (the *active length*, a tunable configuration parameter) controls the amount of memory devoted to caching the file.

Each active client has its own *client cursor* that keeps track of the file offset of the block that was last queued to send to the client. Based on the client cursor positions, we call the leading active client for a file (often the highest-rate client) the *frontrunner* of the file; the trailing clients active on the same file are the *followers*. The frontrunner advances the file cursor according to its current client cursor. This operation refreshes the system cache with new data blocks that are likely also needed by the followers. When the cursor of a follower lags behind the

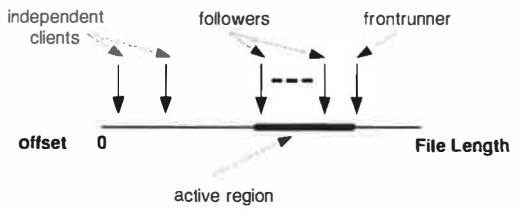


Figure 4: We call the *active region* of a file the part of the file that is resident in server memory. The client who downloads the file with highest rate (*frontrunner*) advances the active region, while clients with lower rates (*followers*) are kept within the active region. Some clients (*independent*) are allowed to move outside the active region.

```
1.  proc circus_algorithm
2.  while (true) do
3.   file := next entry of active_file_circular_list
4.   while (true) do
5.     client := next entry of file.active_client_list
6.     if (client = nil) break
7.     if (client.fifo_queue = full) continue
8.     if (file.cursor-client.cursor > active_length AND
9.       client.sent_blocks_fraction < client.threshold)
10.        client.cursor = file.cursor - active_length/2
11.    client.cursor := next block not transmitted
12.    if (client.cursor = invalid) continue
13.    insert client.cursor location to client.fifo_queue
14.    if (client.cursor > file.cursor AND
15.      client.cursor-file.cursor < leapfront_distance)
16.        file.cursor = client.cursor
17.  done
18. done
```

Figure 5: Pseudocode of the *Circus* block selection algorithm.

frontrunner at a distance that exceeds the active length, the follower's cursor advances to the middle of the active region. Moving the cursor to the middle rather than the front of the active region prevents the follower from immediately becoming a frontrunner. We can summarize these operations with the following two rules:

- Advance the file cursor to the current cursor location of the fastest client (*frontrunner*).
- When the cursor of a trailing client (*follower*) falls behind the *active region*, advance the cursor of the *follower* to the midpoint of the active region.

To avoid stalling clients that are missing only a small number of blocks, *Circus* tracks the fraction of file blocks each client has already received (the *client threshold*). When this fraction exceeds a configured maximum value, the client becomes *independent*; the algorithm selects blocks for independent clients by scanning their block maps for needed blocks, even if they are outside of the active region. A client also becomes independent if it is the frontrunner and its next missing block is more than a configured *leapfront distance* ahead of the current file cursor. This avoids damage to the reference locality of the other active clients. In our experiments, we found a leapfront distance equal to the active length of the file to achieve stable operation in the system. We investigate the sensitivity of the system to several configuration parameters in Section 5.5.

4 Evaluating Circus

This section gives an overview of our methodology for evaluating *Circus*, including the performance metrics of our experiments.

4.1 Prototype Implementation

We incorporated the *Circus* algorithm into a version of the FTP daemon of FreeBSD Release 4.5 (Figure 6). We modified both the FTP client and server to support block reordering using a simple block transfer protocol with sequencing headers.

The current version of the *Circus* server is a fully functional implementation that required no kernel modifications. For each active file the server constructs a header structure (*file header*) containing information about the file. This includes a linked list of headers (*client headers*) for the active clients of each file, and a block FIFO queue of limited length for each active client. The *block size* is a configurable parameter that serves as logical unit of disk and network transfers. When the socket buffer for a client is ready to send, a server process removes a block descriptor from the queue, and submits a request to transfer the block to the network socket. We used the *sendfile()* system call—available in FreeBSD and other operating system kernels—for zero-copy transfers from the disk to the network. A background process runs the block selection algorithm to refill the FIFO queues with descriptors of blocks to transmit to each client. The headers and the queues are maintained in shared memory accessible by all server processes.

The modified FTP client reassembles downloaded files and optionally writes them to the local file system on the client. We disabled the writes in our experiments so that multiple clients (up to a few hundred) could run on each test machine.

4.2 Metrics and Workload

We define *disk throughput* as the total disk bandwidth (byte/s) used by the server, and *network throughput* as the total network bandwidth (byte/s) used to send data over the network to clients (with unicast). We can approximate the *miss ratio* from the ratio of disk throughput

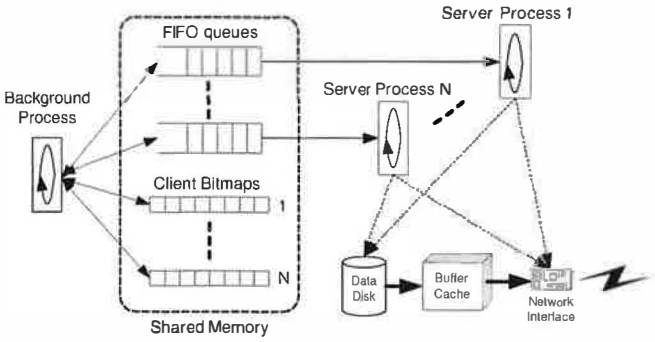


Figure 6: Simplified diagram outlining the prototype implementation of the *Circus* download server. Solid arrows show data transfers, while dotted arrows illustrate transfer of control.

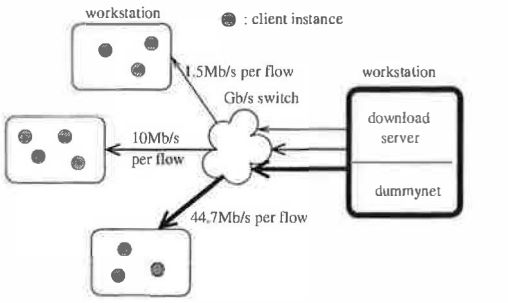


Figure 7: Multiple client instances corresponding to different download requests are partitioned across several workstations according to their network link capacity. The server is connected to the client workstations through a gigabit Ethernet switch. *Dummynet* is used to control the per flow rate from the server to each client workstation.

to the network throughput, ignoring locality effects. A low ratio indicates that disk block fetches are contributing to multiple outstanding transfers for a shared file. The reduced disk activity can improve *download time* and server throughput (the download completion rate).

We assume Poisson arrivals for the download requests because they closely match real workloads studied in previous work [3]. For the definition of the system load ρ during steady-state operation, we take the delivered network throughput as the aggregate transfer rate requested by the users; ideally, each download request should be served at the client's network link rate. Based on the analysis of Section 2, we consider the server's network bandwidth to be the limiting resource: maximum system load $\rho = 1$ occurs when the arrival rate generates network throughput that saturates the server's network link. Depending on the file sizes, this load definition leads to different request arrival rates and different interarrival gaps. We derive the mean request interarrival time corresponding to the maximum system load from the ratio of the average file size over the outgoing server link capacity. For lower loads, the interarrival gap is the ratio of the peak load interarrival time to the load level ρ.

4.3 Measurement Environment

All experiments use Intel PIII-based systems with 256 MB main memory running FreeBSD 4.5 at 733 MHz or 866 MHz. A group of client workstations run multiple client instances to generate request loads to a server. On the server node, we use *Dummynet* [25] to specify the per flow transfer rate from the server node to each client (Figure 7). We store the file data in a 18GB Seagate Cheetah 10K RPM disk with sequential transfer rate 26-40MByte/s. The systems are equipped with both 100 Mbit/s and 1 Gbit/s Ethernet interfaces connected via two separate switches. File network transfers take place over the gigabit switch using jumbo Ethernet packets (9000 bytes) to reduce network protocol overhead.

We focus on handling download requests of files with total storage footprint that exceeds the memory of the server. Most of our experiments use files of size 512MB. Our results also apply when files fit in server memory but the aggregate footprint exceeds server memory. Due to reported correlations between the transferred file size and the client link capacity [31], and to reduce experimentation time, we conservatively consider clients with broadband transfer rates. For each different client link capacity that we support, we dedicate a separate client node and configure its connection speeds to the server through *Dummynet*.

From recent studies in peer-to-peer network systems it has been found that 20-30% of the users have downstream network links less than 1Mbit/s, about 80-90% have downstream links less than 10Mbit/s, and the remaining 10-20% have links that exceed 10Mbit/s [28]. In accordance with the above results and the fact that broadband user population tends to increase over time, we specified three groups of clients with 1.544 Mbit/s (T1), 10 Mbit/s (we call it 10T), and 44.736 Mbit/s (T3). We experiment with each of these groups separately, and also with a mixed workload (Mx) where 20% of the users are of type T1, 60% are of type 10T, and the remaining 20% are type T3. In several cases, we focus on 50%-50% mixes of two client groups. We allow each experiment to run for between $\frac{1}{2}$ hour and $1\frac{1}{2}$ hour, depending on the network link capacity of the clients. Measurements start after one or more initial download requests complete. Each experiment is repeated until the half-length of the 95% confidence interval of the measured network throughput lies within 5% of the estimated mean value across different trials.

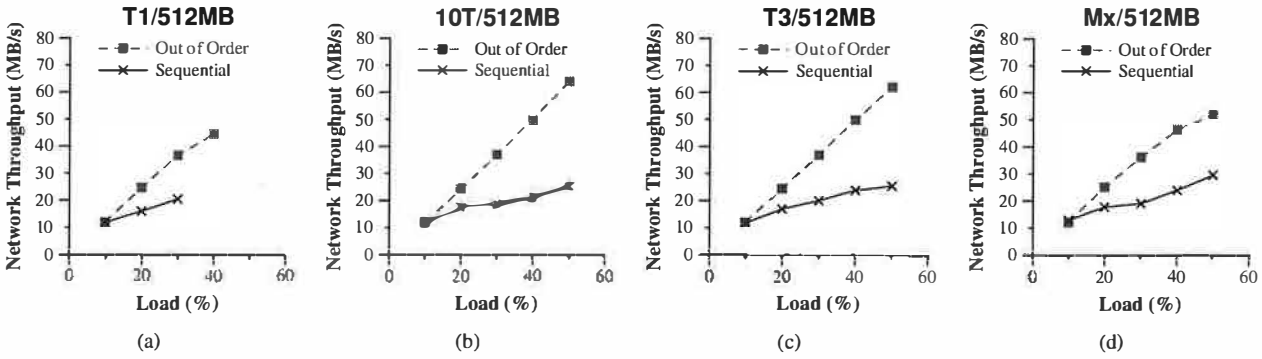


Figure 8: We compare the network throughput of the unmodified (sequential) and *Circus* (out-of-order) download server implementations at increasing system loads using 512MB file requests. We consider the client link capacity to be equal to (a) 1.5Mbit/s, (b) 10Mbit/s, (c) 44.7Mbit/s, and (d) a mix of 20% 1.5Mbit/s, 60% 10Mbit/s and 20% 44.7Mbit/s. The out-of-order approach more than doubles the network throughput at higher loads. The higher the network throughput, the better the system throughput as well.

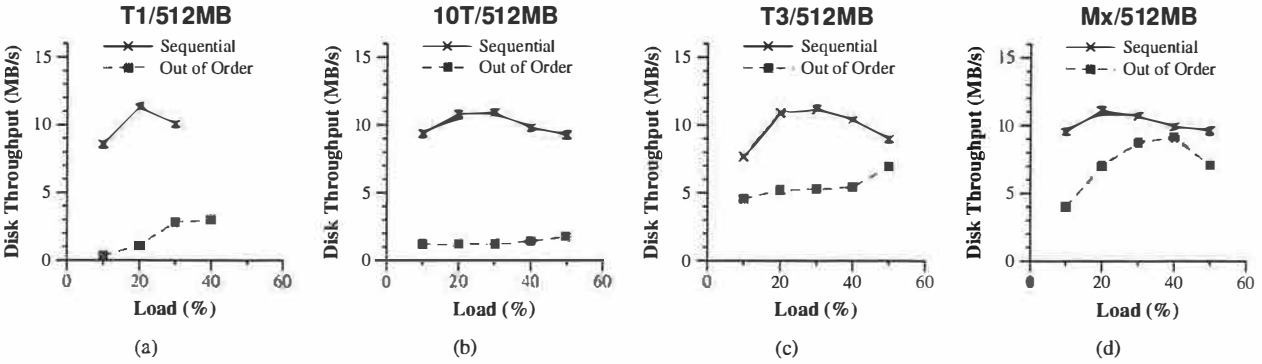


Figure 9: At low and moderate loads, the disk throughput with out-of-order transfers remains roughly equal to the transfer rate of the most demanding individual client across different client link rates (a-d). With sequential transfers, the disk is highly utilized and becomes a bottleneck as shown in Figure 8 (lower disk throughput is better for a given network throughput).

5 Experimental Results

We compare the performance of the *Circus* prototype with an unmodified FreeBSD 4.5 ftpd implementation. The experiments investigate alternative client features, network conditions, file characteristics, and server configuration parameters. We find *Circus* to improve the server throughput and file download time when files are shared by multiple clients.

5.1 Client Link Capacity

A key challenge in the design of a download server is to adapt automatically to different client rates without manual tuning. The closer the transfer rates of two clients match, the easier it becomes to exploit the data sharing among them. As the difference increases, it becomes more difficult to share cached data effectively.

Figure 8 depicts the network throughput of an unmodified (sequential) and a *Circus* (out-of-order) server as clients with different rates download a single file of size 512MB. In typical ftpd implementations (including the one that we use here), each active download request spawns an extra server process with resident memory space of about 1MB. Consequently, we show only T1 measurements for up to 30-40% load, roughly corresponding to about 200 concurrent clients. Beyond this point memory paging interferes with the measurements.

In all the three cases of a single client link rate (a-c), the out-of-order network throughput increases proportionally with the system load. In particular, at 40% load, we expect to receive 51.2MByte/s throughput, which is roughly what we observe in cases (b) and (c). The measured throughput is somewhat lower (in case d) with clients of different rates on the same server, but still reaches 50MByte/s at 50% load. Quite remarkably, the sequential system only matches the out-of-order performance at 10% load in the four cases, and never exceeds 30MByte/s (on average) as the load increases.

Figure 9 shows disk throughput for the same experiment. With sequential transfers, the disk is highly utilized even at low loads, regardless of the client rates. In contrast, with out-of-order transfers (a-c) the disk throughput drops to the transfer rate of a single client. For example the disk throughput is about 1MByte/s with 10T transfers (b), an order of magnitude lower than the se-

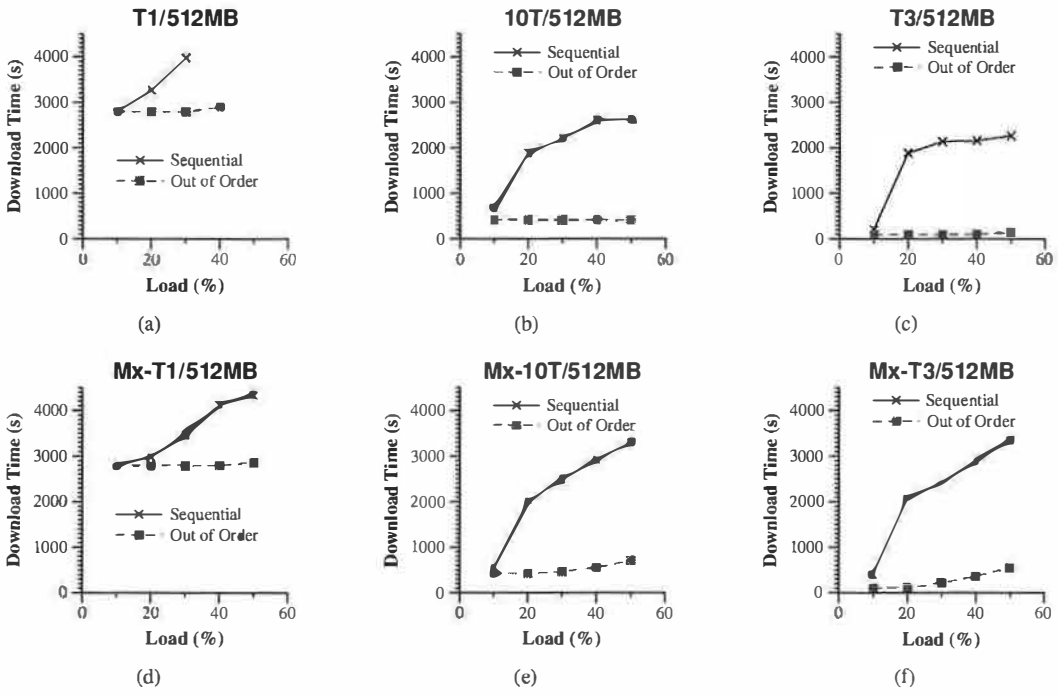


Figure 10: With out-of-order transfers, download requests take almost constant time to complete as the load increases. When file data transfers are served sequentially, however, the download duration increases significantly as a function of the system load. We consider the case where all clients have the same network link capacity (a-c), and when clients of different link capacities are served by a single server (d-f).

quential case. When we mix clients of different capacities (d), this behavior holds at low loads with the disk throughput about 5.6 MByte/s. At higher loads, the proportion of non-sharing (independent) clients increases, raising the disk throughput accordingly. Figure 10 further verifies these observations. With out-of-order transfers, the download latencies remain roughly constant at different system loads, according to the client rates. But when sequential transfers are used, the download latency increases rapidly with the system load.

5.2 Transferred File Size

This section investigates how the file size affects the system performance. Figure 11(a) shows the server network throughput in a file size range between 256MB and 1GB. We observe that, with out-of-order transfers, the network throughput remains above 50MByte/s, consistent with the 40% offered load. Sequential transfers cause the network throughput to drop below 20MByte/s, approaching the disk throughput. As a result, download latency (not shown) increases dramatically for sequential transfers to several tens of minutes. For the out-of-order case all downloads complete within a few minutes at all the file sizes that we examined.

5.3 Multiple Files

Even though it is likely that only a few files will be in heavy demand by the clients, we investigate how the performance of the system is affected when the number of popular files increases. We consider 1 to 16 different files of 512 MB each, all stored on a single server disk, and requested with equal probability. The clients receive data over 10Mbit/s links, and the system is at 40% load. In Figure 12(a), we illustrate the network throughput of the server with sequential and out-of-order transfers respectively. In the out-of-order case, the measured throughput remains roughly 50 MByte/s with up to 8 files, and drops slightly to 48MB/s with 16 files. From Figure 12(b), the average disk throughput increases linearly with the number of files up to eight, and reaches 10MB/s at 16 files. This behavior is expected because the number of disk access streams increases with more active files, and the disk throughput begins to limit the system as it approaches 10MByte/s. With sequential transfers, the disk throughput always limits the system and performance only worsens as the number of files increases.

5.4 Round-trip Delay and Packet Loss

Packet loss rate and propagation delay can vary significantly in a wide-area network depending on the physical span and the operating conditions of the network. We in-

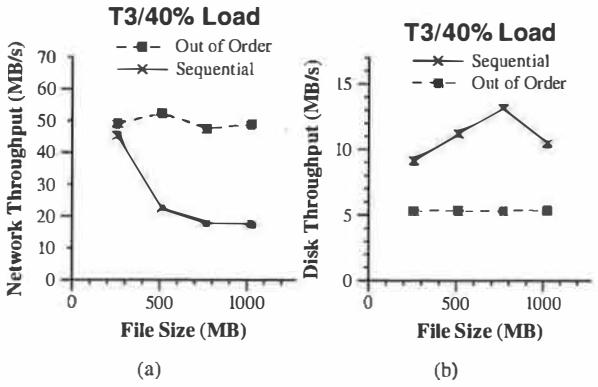


Figure 11: Server network throughput and disk throughput for transfers along T3 links of a file with size between 256MB and 1GB. The system load is set equal to 40%. With out-of-order transfers the the network throughput is always higher and the disk throughput is constant regardless of the size of the file.

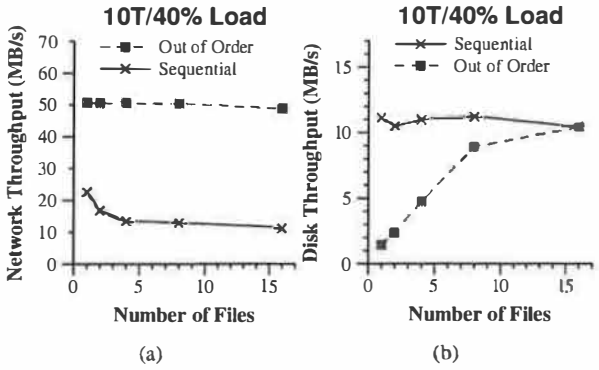


Figure 12: Server network throughput and disk throughput when the number of concurrently requested files varies from 1 to 16. The system load is 40% and the file size is always 512MB. All files are requested with equal probability.

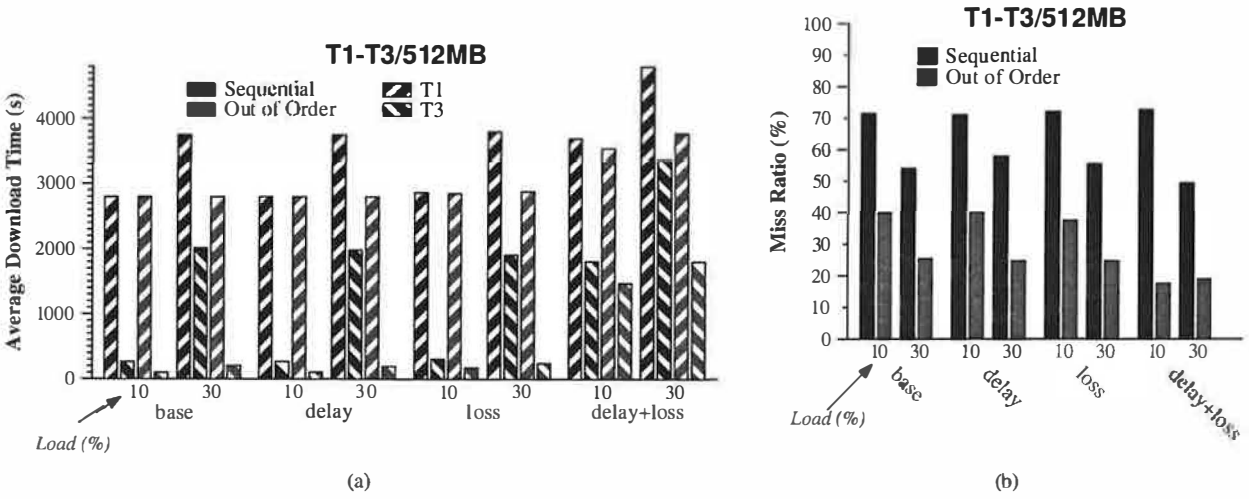


Figure 13: In the base case we assume round-trip delay less than 1ms and packet loss close to 0%. In the delay case we increase the round-trip delay to 75ms and in the loss case we increase the loss rate to 10%, correspondingly, with respect to the base case. Both the download time and the miss ratio of sequential and out-of-order transfers can be affected when combining round-trip delay of 75ms with packet loss rate of 10% (delay+loss). T1 and T3 links are used with equal probability to connect a single server to multiple clients.

vestigated the impact of such factors to file transfers by experimenting with round-trip times of about 1 and 75 ms, and with packet loss rates about 0% and 10%, respectively, using Dummynet. In Figure 13, we measure the download time and server miss ratio when transferring a 512MB file over T1 and T3 links from the same server. When packet loss of 10% and delay of 75ms are combined in out-of-order transfers, download time over T3 links increases by an order of magnitude approaching the level of sequential transfers. This ten-fold increase from the base case can be attributed to the mechanism used by the congestion avoidance algorithm to recover the congestion window at the sender.

Longer round-trip delays increase the recovery time and the wasted network bandwidth. This can be explained by the TCP operation: packet losses lead to triple duplicate acknowledgments (rather than timeouts), and the congestion window increases by at most one data segment every round-trip time [21]. Individual sequential transfers have low throughput due to the disk bottleneck, and are not affected further at low load. However, raising the system load from 10% to 30% doubles the time of T3 sequential transfers, while leaving the out-of-order transfer time almost unchanged. When combining delay and loss with out-of-order transfers, disk throughput drops because data retransmissions hit in the buffer cache. We don't observe similar effects for sequential transfers, which provides additional evidence about the poor disk access locality of this policy.

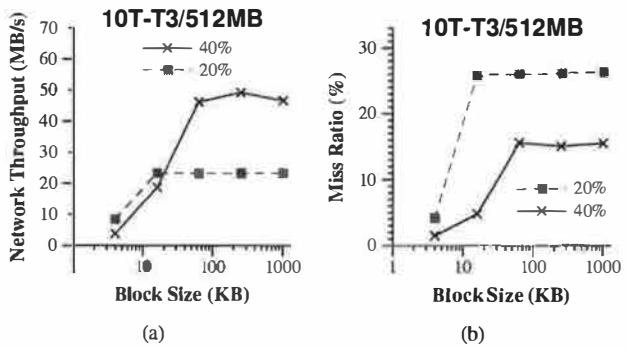


Figure 14: We examine the sensitivity of the system to the block size parameter, when mixing equiprobable requests from clients with 10T and T3 links requesting a file of 512MB at system load 20% and 40%. Both the network throughput and the miss ratio are adversely affected by block sizes smaller than 64KB, but remain insensitive to larger values.

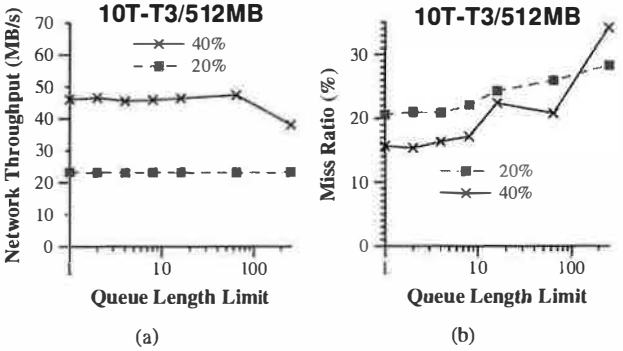


Figure 15: We investigate the effect of the queue length limit, when mixing equiprobable requests from clients with 10T and T3 links for a 512MB file at different loads. As the queue length limit increases, the disk throughput also grows leading to higher miss ratio. Eventually, the disk bandwidth becomes bottleneck which makes the server network throughput to drop.

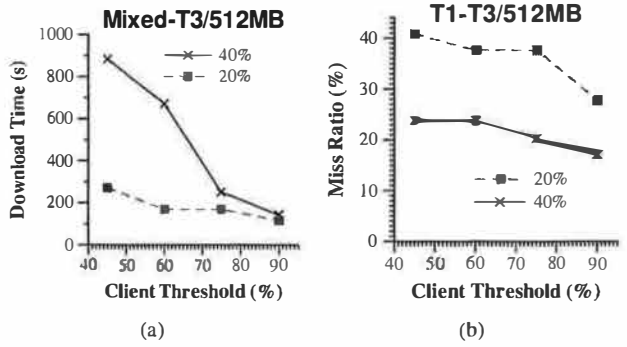


Figure 16: We examine the sensitivity of the system performance to the client threshold when mixing download requests over T1 and T3 network links. We found the client thresholds equal to 75% or higher to keep both the download time over T3 links and the miss ratio low.

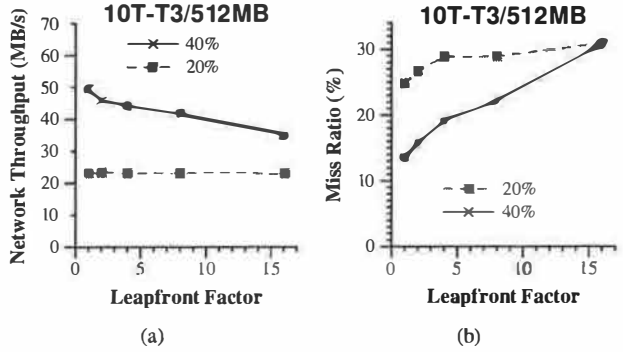


Figure 17: We show the effect of the leapfront factor using equiprobable download requests over 10T and T3 lines at different system loads. As the leapfront factor increases, network throughput drops and miss ratio surges, especially at high system load.

5.5 Sensitivity to System Parameters

This section examines how sensitive the system behavior is to important configuration parameters. We did extensive experimentats to ensure that the system remains robust across a wide range of workloads, but we include only a few representative measurements here. Overall, the system behavior is affected by the configuration parameters below, but remains stable when the parameters remain within the ranges that we suggest.

Block Size. The *block size* is a configurable parameter that specifies the unit of disk access and network transfer requests in the server. Its value affects the utilization of the devices, the overhead involved in the operation of the system, and the overall server throughput. In Figure 14, we illustrate the network throughput and miss ratio across different system loads for block sizes ranging between 4KB and 1MB. We observe that both the measured metrics remain constant with block size larger than 16KB and 64KB at low and high load, respectively. Low loads show higher miss ratios because there is less sharing. Smaller block sizes increase the disk access overhead and block selection overhead. In general, we found the block size equal to 64KB to perform well, and we used it in all the other experiments.

Queue Length Bound. Figure 15 shows the effect of varying the bounds on the block queue FIFOs for active clients. Shorter queues make the system more adaptive to the variability of the client behavior, because the blocks sent to each client are chosen based on recent system conditions. On the other hand, large queue lengths can increase the throughput of the system by keeping each client's network path fully pipelined. We examine the impact of the *queue length limit* on the performance of the system using 512MB download requests over equiprobable 10T and T3 links. With longer queues the miss ratio increases, the disk bandwidth becomes a bottleneck, and the server network throughput drops. This is expected because longer queue lengths can lead to stale requests for blocks that have been evicted from the cache

and incur extra disk activity. In all the other experiments, the queue length limit is set equal to 5.

Client Threshold. The *client threshold* controls the creation of independent clients according to the percentage of file blocks the client has received. From our experiments we found the system to perform well with client threshold around 0.75. Lower client thresholds reduce data sharing, increase disk access activity and lead to longer download duration (Figure 16), while higher client thresholds make the system operation less stable especially with large number of clients.

Leapfront Distance. The *leapfront distance* determines when a client is allowed to play the role of a frontrunner depending on how far ahead from the file cursor the client cursor has moved. For convenience, we introduce the *leapfront factor* as the ratio of the *leapfront distance* over the *active length*. In Figure 17, we notice that as the leapfront factor grows larger than 1, the network throughput drops and the miss ratio increases. Setting the leapfront distance equal to the *active length* gives good performance by allowing the active region to move smoothly forward; larger leapfront distances tend to reduce spatial locality among different clients and lead to lower throughput. The active length was set equal to 16MB throughout our study.

6 Related Work

File Transfer and File Sharing. FTP and HTTP transfer objects sequentially, relying on the TCP transport to preserve byte ordering. With marker blocks [23] it is possible to restart a transmission after a failure. Raman et al. improve the interactive transfer of images over the Internet by delivering data to the client as they arrive, weakening the in-order abstraction of TCP [24]. Diot and Gagnon examine benefits of out-of-sequence packet processing [15], but do not consider large file delivery or interactions with storage devices.

Many wide-area storage systems allow a client to download different parts of a file from multiple servers (e.g., BitTorrent [14]); these clients resequence the data to tolerate out-of-order delivery. Acharya et al. propose a server architecture for repetitive transmission of data over a broadcast channel [1]. The frequency of transmitted data is determined by data popularity across the served client population.

Forward Error Correction. Digital Fountain [9] encodes content with forward error correcting (FEC) codes (e.g., Tornado codes) for distribution over a multicast network. FEC allows a client to reconstruct a file once it has received a minimum number of distinct blocks. This approach eliminates the need for acknowledgments in a multicast setting. The system can be extended to transport large files to a client from multiple collaborating sources in overlay networks [8].

In a unicast network, FEC encoding can be applied to improve caching efficiency at the server [26]. Since the client can reconstruct the data from any sufficiently large subset of the encoded blocks, a block fetched from disk may be useful to multiple clients with different request arrival times and different rates. If a block is lost, another may be sent in its place, avoiding the need for the server to buffer data for retransmission. However, duplicate blocks waste client bandwidth; in a typical heterogeneous environment, where client receiving rates can differ by several orders of magnitude, the encoded version of the transmitted file is much larger than the original to limit the probability that any arbitrary block is a duplicate for some active client [9, 26]. Recent theoretical work begins to address this problem [19]; if a satisfactory solution is found, then FEC could meet our objectives for downloading large files with a high degree of sharing with low network impact, e.g., when multicasting is available.

Circus demonstrates a technique with similar goals for content distribution: to maximize the advantage of data sharing across concurrent requests, while allowing clients at different rates to reassemble the requested file quickly and efficiently. However, *Circus* does not use FEC codes, and it is effective for unicast, although it could also benefit from multicast.

Stream Merging Methods. A class of *merging methods* for multicast delivery of streaming media allows a client to receive data transmitted concurrently to other clients [17, 18]. These file segmentation schemes balance the server network throughput, the client network throughput, and the playback initiation latency. Those schemes are significantly different from *Circus* because they have been specifically designed to support real-time delivery guarantees over reliable multicast-enabled networks assuming a fixed receiving rate for each client. In contrast, *Circus* supports efficient file (or file segment) download transfers over unicast networks for clients with different rates, and exploits the complementary technique of block reordering.

The insights underlying our approach are related to Steere's work with asynchronous set iterators [29], although our approach does not affect the order in which data is delivered to a client application.

7 Conclusions

This paper explores opportunistic block reordering to exploit the data sharing among concurrent file transfers. We introduce the *Circus* algorithm for scheduling disk access

and reordered network transfers, and evaluate an implementation in a modified FTP file server and client under synthetic file access workloads. We conclude that block reordering can significantly improve server cache performance for large, shared files. The average file download time with *Circus* remains close to minimum across the workloads, and is significantly lower than with conventional sequential file download. Additionally, *Circus* more than doubles the server network throughput when there is significant sharing, and reduces the required disk bandwidth by an order of magnitude in some cases.

8 Acknowledgments

We thank Balachander Krishnamurthy for an early discussion, and Amin Vahdat for helpful comments on a draft. Darrell Anderson, Wei Jin, and Ken Yocum also provided useful feedback. Miriam O' Mahony and David Becker offered valuable technical support.

References

[1] Acharya, S., Franklin, M., and Zdonik, S. Balancing Push and Pull for Data Broadcast. In *ACM SIGMOD* (Tuscon, AZ, May 1997), pp. 183–194.

[2] Allcock, B., Bester, J., Bresnahan, J., Chervenak, A. L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnal, D., and Tuecke, S. Data Management and Transfer in High Performance Computational Grid Environments. *Parallel Computing Journal* **28**, 5 (May 2002), 749–771.

[3] Almeida, J. M., Krueger, J., Eager, D. L., and Vernon, M. K. Analysis of Educational Media Server Workloads. In *Intl Workshop on Network and Operating System Support for Digital Audio and Video* (Port Jefferson, NY, June 2001), pp. 21–30.

[4] Anastasiadis, S. V., Sevcik, K. C., and Stumm, M. Modular and Efficient Resource Management in the Exedra Media Server. In *USENIX Symposium on Internet Technologies and Systems* (San Francisco, CA, Mar. 2001), pp. 25–36.

[5] Arlitt, M. F., and Williamson, C. L. Web server workload characterization: the search for invariants. In *ACM SIGMETRICS* (Philadelphia, PA, May 1996), pp. 126–137.

[6] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K. Measurements of a distributed file system. In *ACM Symposium on Operating Systems Principles* (Oct. 1991), pp. 198–212.

[7] Barford, P., and Crovella, M. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *ACM SIGMETRICS* (Madison, WI, July 1998), pp. 151–160.

[8] Byers, J., Considine, J., Mitzenmacher, M., and Rost, S. Informed Content Delivery Across Adaptive Overlay Networks. In *ACM SIGCOMM* (Pittsburgh, PA, Aug. 2002), pp. 47–60.

[9] Byers, J. W., Luby, M., Mitzenmacher, M., and Rege, A. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *ACM SIGCOMM* (Vancouver, BC, Sept. 1998), pp. 57–67.

[10] Cao, P., Felten, E. W., Karlin, A., and Li, K. A Study of Integrated Prefetching and Caching Strategies. In *SIGMETRICS/Peformance '95* (May 1995).

[11] Chesire, M., Wolman, A., Voelker, G. M., and Levy, H. M. Measurement and Analysis of a Streaming-Media Workload. In *USENIX Symposium on Internet Technologies and Systems* (San Francisco, CA, Mar. 2001), pp. 1–12.

[12] Clark, D. D., and Tennenhouse, D. L. Architectural Considerations for a New Generation of Protocols. In *ACM SIGCOMM* (Philadelphia, PA, Sept. 1990), pp. 200–208.

[13] Coffman, K., and Odlyzko, A. M. Internet growth: Is there a "Moore's Law" for data traffic? In *Handbook of Massive Data Sets*. Kluwer Academic, 2002, pp. 47–93.

[14] Cohen, B. Incentives Build Robustness in Bittorrent, May 2003. bitconjurer.org.

[15] Diot, C., and Gagnon, F. Impact of out-of-sequence processing on the performance of data transmission. Computer Networks, 31 (1999), 475–492.

[16] Doyle, R. P., Chase, J. S., Gadde, S., and Vahdat, A. M. The Trickle-Down Effect: Web Caching and Server Request Distribut ion. In *Intl Workshop on Web Caching and Content Delivery* (June 2001).

[17] Eager, D., Vernon, M., and Zahorjan, J. Minimizing Bandwidth Requirements for On-Demand Data Delivery. *IEEE Transactions on Knowledge and Data Engineering* **13**, 5 (September/October 2001), 742–757.

[18] Jin, S., and Bestavros, A. Scalability of Multicast Delivery for Non-sequential Streaming Access. In *ACM SIGMETRICS* (Marina Del Rey, CA, June 2002), pp. 97–107.

[19] Luby, M. LT Codes. In *IEEE Symposium on Foundations of Computer Science* (Vancouver, BC, Nov. 2002), pp. 271–282.

[20] Megiddo, N., and Modha, D. S. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST 03)* (March 2003).

[21] Padhye, J., Firoiu, V., Towsley, D. F., and Kurose, J. F. Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation. *IEEE/ACM Transactions on Networking* **8**, 2 (Apr. 2000), 133–145.

[22] Pai, V. S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. Locality-aware Request Distribution in Cluster-based Network Servers. In *ACM ASPLOS* (San Jose, CA, Oct. 1998), pp. 205–216.

[23] Postel, J., and Reynolds, J. File Transfer Protocol (FTP), Oct. 1985. USC/ISI, Network Working Group RFC 959.

[24] Raman, S., Balakrishnan, H., and Srinivasan, M. An Image Transport Protocol for the Internet. In *Intl Conf. on Network Protocols* (Osaka, Japan, Nov. 2000), pp. 209–219.

[25] Rizzo, L. Dummynet: A simple approach to the evaluation of network protocol. *ACM Communication Review* **47**, 1 (Jan. 1997), 31–41.

[26] Rost, S., Byers, J., and Bestavros, A. The Cyclone Server Architecture: Streamlining Delivery of Popular Content. In *Intl Workshop on Web Caching and Content Distribution* (Boston, MA, June 2001).

[27] Saroiu, S., Gummadi, P. K., Dunn, R. J., Gribble, S. D., and Levy, H. M. An Analysis of Internet Content Delivery Systems. In *USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), pp. 315–328.

[28] Saroiu, S., Gummadi, P. K., and Gribble, S. D. A measurement study of peer-to-peer file sharing systems. In *SPIE/ACM Multimedia Computing and Networking Conference* (Jan. 2002).

[29] Steere, D. C. Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency. In *ACM Symposium on Operating Systems Principles* (Oct. 1997), pp. 252–263.

[30] Wang, L., Pai, V. S., and Peterson, L. L. The Effectiveness of Request Redirection on CDN Robustness. In *USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), pp. 345–360.

[31] Zhang, Y., Breslau, L., Paxson, V., and Shenker, S. On the Characteristics and Origins of Internet Flow Rates. In *ACM SIGCOMM* (Pittsburgh, PA, Aug. 2002).

A framework for building unobtrusive disk maintenance applications

Eno Thereska, Jiri Schindler*, John Bucy, Brandon Salmon,
Christopher R. Lumb, Gregory R. Ganger
Carnegie Mellon University

Abstract

This paper describes a programming framework for clean construction of disk maintenance applications. They can use it to expose the disk activity to be done, and then process completed requests as they are reported. The system ensures that these applications make steady forward progress without competing for disk access with a system's primary applications. It opportunistically completes maintenance requests by using disk idle time and freeblock scheduling. In this paper, three disk maintenance applications (backup, write-back cache destaging, and disk layout reorganization) are adapted to the system support and evaluated on a FreeBSD implementation. All are shown to successfully execute in busy systems with minimal (e.g., <2%) impact on foreground disk performance. In fact, by modifying FreeBSD's cache to write dirty blocks for free, the average read cache miss response time is decreased by 15–30%. For non-volatile caches, the reduction is almost 50%.

1 Introduction

There are many disk maintenance activities that are required for robust system operation and, yet, have loose time constraints. Such "background" activities need to complete within a reasonable amount of time, but are generally intended to occur during otherwise idle time so as to not interfere with higher-priority application progress. Examples include write-back cache flushing, defragmentation, backup, integrity checking, virus scanning, report generation, tamper detection, and index generation.

Current systems use a variety of ad hoc approaches for such activities. Most trickle small amounts of work into the storage subsystem, either periodically or when an idle period is detected. When sufficient idle time is not available, these activities either compete with foreground requests or are not completed. More importantly, trickling work into a storage subsystem wastes significant disk scheduling opportunities—it restricts the scheduler to only considering a small subset of externally-chosen requests at externally-chosen points in time. Most background activities need to read or write substantial portions of the disk, but do not have particular ordering requirements. As a result, some implementors try hard to initiate the right requests at the right times, introducing substantial complexity but, usually, only minor improvement.

This paper describes an alternate approach, wherein background activities are exposed to the storage subsystem so that it can schedule associated disk accesses opportunistically. With the storage subsystem explicitly supporting priorities, background applications can safely expose work and trust that it will not interfere with foreground activity. Doing so allows the scheduler to use freeblock scheduling and idle disk time to complete background disk accesses in the most device-efficient manner. Freeblock scheduling [21] predicts rotational latency delays and tries to fill them with media transfers for background tasks. As the set of desired disk locations grows, so does the ability of a freeblock scheduler to utilize such latency delays. The same is true for non-intrusive use of short periods of idle time. Combining rotational latency gaps with short and long periods of idle time, programs designed to work with storage-determined ordering can make consistent progress, without affecting foreground access times, across a wide range of workloads and levels of activity.

This paper describes a framework for background disk activities, including application programming interfaces (APIs) and support for them in FreeBSD. In-kernel and system call APIs allow background applications to register "freeblock tasks." Our *freeblock subsystem* replaces the generic SCSI driver's disk scheduler, utilizing both freeblock scheduling and any idle time to opportunistically complete freeblock requests. The APIs are explicitly asynchronous, and they encourage implementors to expose as much background work as possible. For example, dynamic buffer management allows freeblock tasks to register a desire to read more disk space than fits in main memory. Just-in-time locking avoids excessive holding of buffers, since freeblock writes may be pending for a long time. Rate control avoids memory exhaustion and wasted disk scheduling efforts for applications without sufficient CPU time or network bandwidth.

We describe the conversion of three disk maintenance tasks to use this infrastructure: scanning of disk contents for backup, flushing of write-back caches, and reorganizing disk layouts. Well-managed systems perform pe-

*Currently with EMC Corporation

riodic backups, preferably without interfering with foreground activity. Backup is an excellent match for our framework, often reading large fractions of the disk; a "physical" backup does so without interpreting the file system structures and can be made order-agnostic. We implemented such a physical backup application that uses the freeblock subsystem to read disk blocks. Physical backup of a snapshot that covers 70% of an always-busy 18GB disk can be completed in a little over one hour with less than 2% impact on a foreground workload.

Almost all file servers and disk array controllers use write-back caching to achieve acceptable performance. Once updates are decoupled from application progress, they become a background activity appropriate for our framework. In our evaluations, we find that approximately 80% of the cache flushes can usually be eliminated, even when there is no idle time, reducing the average disk read response time by 12-25%. For low read-write ratios (e.g., 1:3–1:1), only 30–55% of the flushes are eliminated, but the read response time reductions are still 15–30%. Interestingly, when emulating a non-volatile cache, which eliminates FreeBSD's 30-second limit on time before write-back, almost all flushes can be eliminated, improving read response times by almost 50%.

Over time, allocated storage space becomes fragmented, creating a desire for defragmentation. Also, there have been many proposals for periodically reorganizing disk layouts to reduce future access times. Both require that disk blocks be shuffled to conform to a new, preferred layout. Our evaluations show that, using our framework, it is possible to reorganize layouts quickly and with minimal impact on foreground workloads.

The remainder of this paper is organized as follows. Section 2 discusses disk maintenance tasks, freeblock scheduling, and related work. Section 3 describes in-kernel and application-level APIs for background disk tasks. Section 4 describes three disk maintenance applications and how they use the APIs. Section 5 briefly describes the freeblock subsystem and its integration into FreeBSD. Section 6 evaluates how well the three applications work when using the framework.

2 Background and related work

There are many disk maintenance activities that need to eventually complete, but that ideally progress without affecting the performance of foreground activity. This section describes how such activities are commonly implemented, how a freeblock subsystem can help, and related work.

Characteristics and approaches: Disk maintenance activities generally have long time horizons for completion, allowing them to have lower priority at any instant than other applications running on a system. As a result, one common approach is to simply postpone such activities until expected off hours; for example, desktop backups are usually scheduled for late nights (or, early in the morning for CS researchers). For less sporadically-used systems, however, the lower priority must be handled in another way.

Another common approach is to spread background requests over time so as to reduce interference with foreground work; for example, some caches flush a fraction of the dirty blocks each second to reduce penalties associated with periodic full cache flushes [6]. More aggressive implementations explicitly identify periods of idle time and use them to service background work. Of course, identifying idle times requires effort—the background activity must be invoked in the system's critical path—and assumptions about any proactive storage-internal functions. When using a detected idle period, background activities usually provide only a few requests at a time to the storage subsystem to avoid having a lengthy queue when the next foreground request arrives. This is necessary because current storage systems provide little-to-no support for request priorities or abort.

By providing only a few requests at a time, these implementations rob the disk scheduler of opportunities to reduce positioning times. In fact, disk maintenance applications usually need to access many disk locations, and many could be quite flexible in their operation ordering. Some implementors attempt to recapture at least a portion of the lost efficiency by providing requests expected to be fast; for example, a disk array reconstruction task can, after a foreground request completes, generate background requests for locations near the recent foreground request rather than near the most recent background request [15]. Such tricks can provide marginal gains, but still lose out on much of the opportunity and often increase complexity by breaking abstraction boundaries between the application and the disk.

Freeblock scheduling: Since disk platters rotate continuously, a given sector will reach the disk head at a given time independent of what the disk head is doing until that time. Freeblock scheduling [21] consists of squeezing background media transfers into foreground rotational latencies. A freeblock scheduler predicts how much rotational latency would occur before the next foreground media transfer and inserts additional media transfers, while still leaving time for the disk head to reach the destination track in time for the foreground transfer. The additional media transfers may be on the current or destination tracks, on another track near the two, or anywhere between them. In the two latter cases, additional seek overheads are incurred, reducing the time available for the additional media transfers, but

not completely eliminating it.

Freeblock scheduling, as originally proposed, combines nicely with idle time usage to provide disk bandwidth to background tasks across a wide range of foreground usage patterns. In addition to detecting and using lengthy idle time periods, low-level scheduling can allow short, sporadic idle periods to be used with minimal penalty. Throughout this paper, we use the term *freeblock scheduling* to refer to this more complete combination of the scheduler that works well when the system is busy with the scheduler that utilizes idle time.

Freeblock scheduling is a good match for many disk maintenance activities, which desire large numbers of disk blocks without requiring a predetermined order of access. Properly implemented, such activities can provide much freedom to a scheduler that opportunistically matches rotational latency gaps and idle time bursts to desired background transfers.

Related work: Lumb et al. [21] coined the term "freeblock scheduling" and evaluated the use of rotational latency gaps for background work via simulation. The simulations indicated that 20–50% of a never-idle disk's bandwidth could be provided to background applications with no effect on foreground response times. This bandwidth was shown to be more than enough for free segment cleaning in a log-structured file system or for free disk scrubbing in a transaction processing system.

Later work by two groups [20, 33] demonstrated that outside-the-disk freeblock scheduling works,[1] albeit with more than 35% loss in efficiency when compared to the hypothetical inside-the-disk implementation assumed in Lumb et al.'s simulations. In both cases, the freeblock scheduler was tailored to a particular application, either background disk scans [20] or writes in eager writing disk arrays [33]. In both cases, evaluation was based on I/O traces or synthetic workloads, because system integration was secondary to the main contribution: demonstrating and evaluating the scheduler. This paper builds on this prior work by describing a general programming framework for background disk tasks and evaluating several uses of it.

Several interfaces have been devised to allow application writers to expose asynchronous and order-independent access patterns to storage systems. Dynamic sets [28], disk-directed I/O [19], and River [2] all provide such interfaces. We borrow from these, and asynchronous networking interfaces like sockets, for the APIs described in the next section.

There has been much research on priority-based scheduling of system resources. Most focus on ensuring that higher priority tasks get the available resources. Notably, MS Manners [9] provides a framework for regulating applications that compete for system resources. Such system support is orthogonal to the framework described here, which creates and maximizes opportunities for progress on background disk accesses. More closely related to freeblock scheduling are real-time disk schedulers that use slack in deadlines to service non-real-time requests [3, 23, 27]; the main difference is that foreground requests have no deadlines other than "ASAP", so the "slack" exists only in rotational latency gaps or idle time.

[1] Freeblock scheduling can only be done at a place that sees the raw disk. So, it could be done within a software logical volume manager but not above a disk array controller. Inside the array controller would work.

3 Background disk I/O interfaces

To work well with freeblock scheduling, applications must be designed explicitly for asynchronous I/O and minimal ordering requirements. An application should describe to the freeblock subsystem sets of disk locations that they want to read or write. Internally, when it can, the freeblock subsystem inserts requests into the sequence sent to the disk. After each desired location is accessed, in whatever order the freeblock subsystem chooses, the application is informed and given any data read.

This section describes two generic application APIs for background activities. The first is an in-kernel API intended to be the lowest interface before requests are sent to the storage device. The second API specifies system calls that allow user-level applications to tap into a freeblock subsystem. These APIs provide a clean mechanism for registering background disk requests and processing them as they complete. Applications written to these interfaces work well across a range of foreground usage patterns, from always-busy to frequently-idle. Both APIs talk in terms of logical block numbers (LBNs) within a storage logical unit (LUN); consequences of this choice are discussed in Section 3.4.

3.1 In-kernel API

Table 1 shows the in-kernel API calls for our freeblock scheduling subsystem. It includes calls for registering read and write freeblock tasks, for aborting and promoting registered tasks, and for suspending and resuming registered tasks. As a part of the high-level device driver, there is one instance of the freeblock scheduler per device in the system; the standard driver call switch mechanism disambiguates which device is intended. This section explains important characteristics of the API.

Applications begin an interaction with the freeblock subsystem with *fb_open*, which creates a *freeblock session*. *fb_read* and *fb_write* are used to add *freeblock tasks*, registering interest in reading or writing specific

Function Name	Arguments	Description
fb_open	*priority, callback_fn, getbuffer_fn*	Open a freeblock session (ret: *session_id*)
fb_close	*session_id*	Close a freeblock session
fb_read	*session_id, addr_range, blksize, callback_param*	Register a freeblock read task
fb_write	*session_id, addr_range, blksize, callback_param*	Register a freeblock write task
fb_abort	*session_id, addr_range*	Abort parts of registered tasks
fb_promote	*session_id, addr_range*	Promote parts of registered tasks
fb_suspend	*session_id*	Suspend scheduling of a session's tasks
fb_resume	*session_id*	Resume scheduling of a session's tasks
**(callback_fn)*	*session_id, addr, buffer, flags, callback_param*	Report that part of task completed
**(getbuffer_fn)*	*session_id, addr, callback_param*	Get memory address for selected write

Table 1: **In-kernel interface to the freeblock subsystem.** *fb_open* and *fb_close* open and close a freeblock session for an application. Tasks can be added to a session until the application closes it. *fb_read* and *fb_write* register one or more freeblock tasks. *fb_abort* and *fb_promote* are applied to previously registered tasks, to either cancel pending freeblock tasks or convert them to foreground requests. *fb_suspend* and *fb_resume* disable and enable scheduling for all tasks of the specified session. **(callback_fn)* is called by the freeblock subsystem to report data availability (or just completion) of a read (or write) task. When a write subtask is selected by the scheduler, **(getbuffer_fn)* is called to get the source memory address.

disk locations, to an open session.[2] Sessions allow applications to suspend, resume, and set priorities (values between 1 and 100, with a default of 20) on collections of tasks.

No call into the freeblock scheduling subsystem waits for a disk access. Calls to register freeblock tasks return after initializing data structures, and subsequent callbacks report subtask completions. The freeblock subsystem promises to read or write each identified disk location once and to call *callback_fn* when freeblock requests complete. On the last callback for a given session, the *flags* value is set to the value indicating completion.

Each task has an associated *blksize*, which is the unit of data (aligned relative to the first address requested) to be returned in each *callback_fn* call. This parameter of task registration exists to ensure that reads and writes are done in units useful to the application, such as file system blocks or database pages. Having only a portion of a database page, for example, may be insufficient to process the records therein. The *blksize* value must be a multiple of the LBN size (usually 512 bytes). In practice, high *blksize* values (e.g., > 64KB for the disks used in our work) reduce the scheduler's effectiveness.

Calls to register freeblock tasks can specify memory locations, in the *addr_range* structure, but they are not expected to do so. If they don't, for reads, the freeblock scheduling subsystem passes back, as a parameter to *callback_fn*, pointers to buffers that are part of the general memory pool; no memory copies are involved, and the application releases them when appropriate. For writes, the associated *getbuffer_fn* is called when the freeblock scheduler selects a part of a write task. The *getbuffer_fn* either returns a pointer to the memory locations to be written or indicates that the write cannot currently be performed.

The original reason for *getbuffer_fn* was to avoid long-held locks on buffers associated with registered freeblock write tasks. Commonly, file systems and database systems lock cache blocks for which disk writes are outstanding to prevent them from being updated while being DMA'd to storage. With freeblock scheduling, writes can be waiting to be scheduled for a long time; such locks could easily be a system bottleneck. The *getbuffer_fn* callback allows the lock to be acquired at the last moment and held only for the duration of the actual disk write. For example, the free write-backs described in Section 4.2 actually hurt performance when they do not utilize this functionality. Since adding it to the API, we have found that the *getbuffer_fn* function cleanly supports other uses. For example, it enables a form of eager writing [11, 31]: one can register freeblock write tasks for a collection of unallocated disk locations and bind unwritten new blocks to locations in *getbuffer_fn*. The disk write then occurs for free, and the relevant metadata can be updated with the resulting location.

The non-blocking and non-ordered nature of the interface is tailored to match freeblock scheduling's nature. Other aspects of the interface help applications increase the set of blocks asked for at once. Late-binding of memory buffers allows registration of larger freeblock tasks than memory resources would otherwise allow. For example, disk scanning tasks can simply ask for all blocks on the disk in one freeblock task. The *fb_abort* call allows task registration for more data than are absolutely required (e.g., a search that only needs one match). The *fb_promote* call allows one to convert freeblock tasks that may soon impact foreground appli-

[2] The term *freeblock request* is purposefully being avoided in the API to avoid confusion with disk accesses scheduled inside the freeblock subsystem.

cation performance (e.g., a space compression task that has not made sufficient progress) to foreground requests. The *fb_suspend* and *fb_resume* calls allow registration of many tasks even when result processing sometimes requires flow control on their completion rate.

3.2 Application-level API

The application-level API mirrors the in-kernel API, with a system call for each *fb_xxx* function call. The main differences are in notification and memory management. Because the kernel must protect itself from misbehaving applications, the simple callback mechanisms of the low-level API are not feasible in most systems. Instead, a socket-like interface is used for both.

As with the in-kernel API, an application begins by calling *sys_fb_open* to get a *session_id*. It can then register freeblock tasks within the session. For each block read or written via these tasks, a completion record is inserted into buffers associated with the session. Applications get records from these buffers via the one new call: *sys_fb_getrecord (buffer)*; each call copies one record into the specified application *buffer*. Each record contains the *session*, *addr* and *flags* fields from *callback_fn* in the in-kernel API, as well as the data in the case of freeblock reads. Note that a copy is required from the in-kernel buffer to the application layer. An alternate interface, such as that used by IO-lite [24], could eliminate such copies. Like with sockets, the *sys_fb_getrecord* call can be used for both blocking and polling programming styles.[3] A *timeout* parameter in the *sys_fb_getrecord* function dictates how long the application will wait if no completion record is currently available. A value of 0 will return immediately (polling), and a value of -1 will wait indefinitely.

3.3 Consistency model

Freeblock tasks may have long durations; for example, a background disk scan can take over an hour. Therefore, a clear consistency model is needed for overlapping concurrent freeblock and foreground requests.

Like most low-level storage interfaces, our APIs opt for maximum scheduling flexibility by enforcing a minimalistic semantic with three rules. First, no ordering guarantees are enforced among pending tasks, whether they overlap or not. As with traditional I/O interfaces, applications must deal with ordering restrictions explicitly [12]. Second, data returned from a read should have been on the disk media at some point before being returned. Third, a block write can be reported complete when it is on disk or when a concurrent write to the same disk location completes; the latter case is rationalized by the fact that the non-written blocks could have been put on the disk just before the ones actually put there.

Given these semantics, a freeblock scheduler can coalesce some overlapping tasks. Of course, data fetched from media can be replicated in memory and passed to all concurrent readers. In addition, completion of a write task to location A allows completion of all pending reads or writes to A because the newly written data will be the current on-disk data once the write completes. As a result, a write is given preference when a set of overlapping reads and writes are pending; a read could be done before the write, but doing so is unnecessary given the consistency model. Note that completing reads in this way requires that applications not be allowed to update the source RAM during the write, since it is impossible to know when the DMA happened in most systems. Alternately, this enhancement could be disabled, as we have observed little benefit from it in practice.

3.4 Consequences of LBN-based interfaces

The freeblock scheduling APIs described interact with driver-level scheduling in terms of LBNs. This simplifies implementation of the scheduler and of low-level disk maintenance tasks, such as RAID scrubbing and physical backup. But, many utilities that access structured storage (e.g., files or databases) must coordinate in some way with the software components that provide that structure. For example, consider a file-based backup application. It could read a directory and register freeblock tasks to fetch the files in it, but it will not know whether any given file is deleted and its inode reallocated between the task being registered and the inode eventually being read from disk. If this happens, the application will backup the new file under the old name. Worse problems can arise when directory or indirect blocks are reallocated for file data.

Three options exist for maintenance tasks that interact with structured storage. First, the task could coordinate explicitly with the file system or database system. Such coordination can be straightforward for integrated activities, such as segment cleaning in a log-structured file system, or index generation in a database system. The write-back support in Section 4.2 is an example of this approach. Second, the task could insist that the file system or database system be temporarily halted, such as by `unmount`ing the file system. Although heavy-handed, a system with many file systems could have individual ones halted and processed one-by-one while the others continue to operate on the storage devices. Third, the task could take advantage of an increasingly common

[3] Our experiences indicate that full integration with existing system call mechanisms would be appropriate. Specifically, using the standard file descriptor mechanism would allow integrated use of **`select()`** with sockets, from which this interface borrows many characteristics. For example, given such integration, an application could cleanly wait for any of a set of sockets and freeblock sessions to have made progress.

mechanism in storage systems: the snapshot [14]. A snapshot provides an instance of a dataset as it was at a point in history, which is useful for backup [7] and remote replication [25]. Since the contents of a snapshot remain static, update problems are not an issue for tasks using the freeblock scheduling APIs. In addition to traditional backup tasks, snapshots offer a convenient loose coordination mechanism for disk maintenance tasks like integrity checking, virus scanning, report generation, tamper detection, and garbage collection. Section 4.1 describes an example of how a backup application interacts with the snapshot system and the freeblock subsystem.

The LBN-based interface also bypasses any file-level protections. So, applications using it must have read-only (for read-only activity) or read/write permissions to the entire partition being accessed. Fortunately, most disk maintenance applications satisfy this requirement.

4 Example applications

Many disk maintenance applications can be converted to the programming model embodied in our APIs. This section describes the conversion of three such applications and discusses insights gained from doing so. These insights should help with designing other maintenance applications to use the framework.

4.1 Snapshot-based backup

Most systems are periodically backed-up to ensure that the data stored is not lost by user error or system corruption. In general, it is accepted that either the system will be otherwise idle during the backup time or the backup will have significant performance impact on foreground activity [10, 17].

Backup strategies fall into two categories: logical and physical backup. Logical backup is a file-based strategy. It first needs to interpret the file system's metadata and find the files that need to be backed-up. The files are then stored to the backup media in a canonical representation that can be restored at a later time. The advantages of logical backup include the ability to restore specific files and to backup only live data. Physical backup is a block-based strategy. Physical backup does not interpret the file structure that it is backing up. Uninterpreted raw blocks are copied from one media to another. The main advantages of physical backup are its simplicity and scalability. In particular, physical backup can achieve much higher throughput while consuming less CPU [17].

Physical backup fits well with our programming model. No ordering among blocks is required. Instead, blocks are copied from one device to another as they are read. The blocks could be written to the backup media out of order (and reorganized during restore), or a staging device could be used to reorder before writing to tape. Physical backup can also take advantage of snapshots, which allow consistent backup from an active online system.

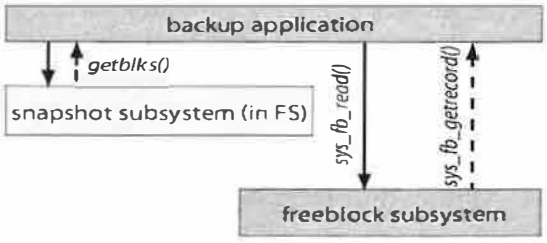


Figure 1: **Snapshot-based backup application.** The backup application interacts with the snapshot subsystem to learn which blocks comprise the snapshot in question. It uses the freeblock subsystem to read these blocks from disk.

Our backup application uses FreeBSD 5.x's snapshot infrastructure [22] and our system call API. No changes are required to the FreeBSD snapshot implementation. After a snapshot is taken, the backup application interacts with the snapshot subsystem as shown in Figure 1. First, it gets the list of blocks that belong to the snapshot file. Then, the backup application registers freeblock tasks, via *sys_fb_read*, to read them. It interactively calls *sys_fb_getrecord* to wait for reads to complete and get the data. Each successfully read block is sent to the back-up destination, together with its address. The backup application can also be used to create a replica by writing each block directly to the corresponding LBN on the destination LUN.

FreeBSD's approach to handling modifications to blocks "owned" by a snapshot creates an additional complexity for the backup application. A snapshot implementation can do one of two things when a block is modified. In the first option ("application-copy-on-write"), a new location is chosen for the updated blocks and the snapshot map stays unchanged. Network Appliance's WAFL file system, for example, uses this method [17]. In the second option ("snapshot-copy-on-write"), the original data is copied to a newly allocated block and the snapshot map is modified. FreeBSD uses this second option to avoid disrupting carefully chosen disk assignments. In the evaluation section, we explore the effects of both methods on the backup application.

To handle FreeBSD's snapshot-copy-on-write, the backup application needs to check with the snapshot system whether each returned block still has the original desired contents. If not, a new freeblock task to read the relocated block is registered. This procedure continues until all original blocks have been read. Note that we could have changed the snapshot subsystem to automatically abort and re-register tasks for modified blocks, but our intention is to show that the backup application

works well even with an unmodified snapshot system.

4.2 Buffer cache cleaner

Caches are part of all storage systems, and most are write-back in nature. Data blocks written to the cache are marked *dirty* and must eventually make their way to the storage device. In most operating systems, including FreeBSD, the cache manager promises applications that data written to the cache will propagate to persistent storage within a certain fixed window of time, often 30 seconds. This persistence policy tries to bound the amount of lost work in the face of a system crash. In many file servers and disk array controllers, cache persistence is not a concern because they utilize battery-backed RAM or NVRAM. But, dirty buffers must still be written to storage devices to make room in the cache for new data. Although these systems do not necessarily need a persistence policy, they still need a cache write-back replacement policy.

Cache write-back is a good application for a freeblock subsystem. In most cases, there are no ordering requirements and no immediate-term timeline requirements for dirty blocks. Until a persistence policy or cache space exhaustion is triggered, write-backs are background activities that should not interfere with foreground disk accesses (e.g., cache misses).

We modified FreeBSD's cache manager to utilize our in-kernel API. It registers all dirty buffers to be written for free through the use of the *fb_write* call. Importantly, cache blocks are not locked when the writes are registered; when its *getbuffer_fn* is called by the freeblock subsystem, the cache manager returns **NULL** if the lock is not free. When its *callback_fn* is called, the cache manager marks the associated block as clean. If the freeblock subsystem still has not written a buffer for free when the cache manager decides it must be written (as a consequence of cache replacement or persistence policies), then the cache manager converts the associated write to a foreground request via *fb_promote*. If a dirty buffer dies in cache, for example because it is part of a deleted file, the task registered to flush it to disk is aborted through *fb_abort*.

4.3 Layout reorganizer

Disk access times are usually dominated by positioning times. Various layout reorganization heuristics have been developed to reduce access times. For example, blocks or files may be rearranged in an organ pipe fashion, or replicated so each read can access the closest replica [16, 32].

Layout reorganization is a background activity that can be made to fit our programming model. But, doing so requires that the implementer think differently about the problem. In the traditional approach, most work focuses on planning an optimal movement pattern. Because a freeblock subsystem is opportunistic, extensive forward planning is not useful, since one cannot predict which freeblock opportunities will be available when.

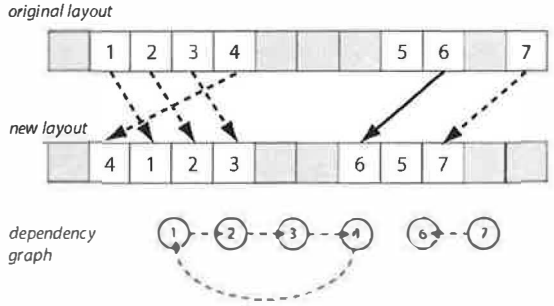


Figure 2: **Sample dependency graph for disk layout reorganization.** This diagram illustrates the dependency graph that results from changing the disk layout. The gray boxes represent empty physical locations. The white boxes present physical locations that have been mapped to a particular block (identified by the number on the box). Dashed arrows present dependencies whereas solid lines show movements that do not have any dependencies.

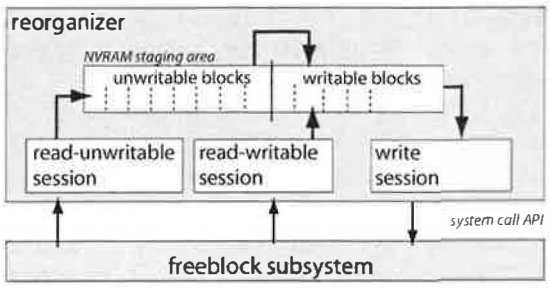


Figure 3: **Layout reorganizer architecture.** This diagram illustrates the design of the layout reorganizer implemented using our framework. The read-unwritable session manages blocks whose dependencies have not yet been solved. The read-writable session manages all blocks that can be read because their dependencies have been solved. The write session manages all block writes. All data is temporarily stored in the NVRAM staging area.

Planning disk reorganization is difficult because there are dependencies between block movements. If a block is to be moved to a location that currently contains live data, the live data must first be read and either moved or buffered. Although no block can directly depend on more than one other block, dependency chains can be arbitrarily deep or cyclic. Figure 2 illustrates an example of these dependencies.

The reorganization module can break a dependency by reading a block into an NVRAM[4] staging area; once the data has been read from a location, new data can safely be written to that location. However, the reorga-

[4]Our experimental system does not actually have NVRAM. Instead, the layout reorganizer just allocates a block of memory and pretends it is non-volatile. This emulates how the reorganizer might work in many modern file servers and disk array controllers.

nizer can still deadlock if it fills the staging area with blocks that cannot be written to disk because of unresolved dependencies. The goal of the reorganizer is to allow the freeblock system maximum flexibility while avoiding this deadlock case. To accomplish this objective, our reorganizer, illustrated in Figure 3, logically partitions the staging area into two two parts: writable and unwritable.

The reorganizer uses three freeblock sessions to move blocks. The "read-unwritable" session registers read tasks for all blocks that cannot yet be written, due to a dependency. The "read-writable" session registers read tasks for blocks that can either be immediately written after they are read (i.e., they have no dependencies) or that clear a dependency for a currently buffered block.

When a read completes, it may eliminate the dependency of another block. If a read-unwritable task is scheduled for this dependent block, the read-unwritable task is aborted (*sys_fb_abort*) and re-registered as a read-writable task. If the dependent block is already in the staging area, it will be changed from an unwritable block to a writable block. A write is scheduled in the "write" session for each writable block in the staging area. When a write completes, its buffer can be released from the staging area and reclaimed.

In order to avoid deadlocking, the reorganizer ensures that the number of unwritable blocks in the cache never exceeds a threshold percentage of the cache.[5] If the number of unwritable blocks reaches the threshold, the reorganizer suspends (*sys_fb_suspend*) the read-unwritable session. However, the read-writable session cannot increase the number of unwritable blocks in the staging area, and can be allowed to continue. When the number of unwritable blocks falls below the threshold, because of writes and/or cleared dependencies, the read-unwritable session can be restarted via *sys_fb_resume*.

The reorganizer must suspend both read sessions when the staging area is filled. However, it cannot deadlock because the reorganizer limits the number of unwritable blocks in the staging area, thus assuring that some number of the blocks in the staging area are writable. The reorganizer simply waits until enough of these writable blocks are written out to disk before resuming the read sessions.

5 The freeblock subsystem

This section briefly describes the freeblock subsystem implemented in FreeBSD to experiment with our background applications. This infrastructure supports all the background disk I/O APIs described in Section 3. Details and evaluation of this infrastructure are available in [29].

[5] The threshold we use is 50%.

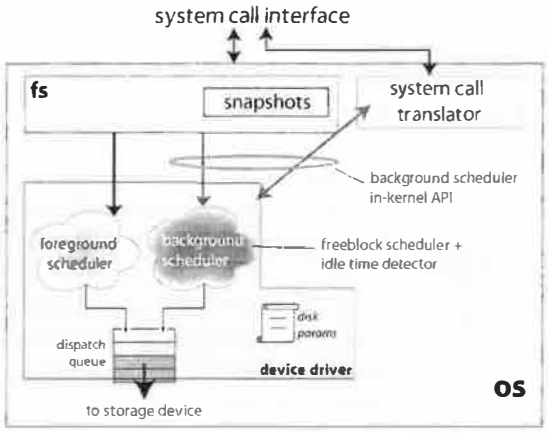


Figure 4: **Freeblock subsystem components.**

5.1 Architecture and integration

Figure 4 shows the major components of our freeblock subsystem. The background scheduler exports the in-kernel API, and a *system call translator* component translates the application-level API calls to in-kernel calls. This section describes these pieces and their integration into FreeBSD.

Foreground and background schedulers: Our scheduling infrastructure replaces FreeBSD's C-LOOK scheduler. The foreground scheduler uses Shortest-Positioning-Time-First (SPTF), and the background scheduler uses freeblock scheduling (rotational latency gaps and any idle time). Both schedulers use common library functions, based on Lumb et al.'s software-only outside-the-disk SPTF models [20], for modeling the disk to predict positioning times for requests.

Like the original scheduler, our foreground scheduler is called from FreeBSD's *dastrategy()* function. When invoked, the foreground scheduler appends a request onto the driver's device queue, *buf_queue*, which is the dispatch queue in Figure 4. It then invokes the background scheduler, which may create and insert one or more freeblock requests ahead of the new foreground request.

When a disk request completes at the disk, FreeBSD's *dadone()* function is called. Into this function, we inserted calls to the background and foreground schedulers. The background scheduler code determines whether the completed request satisfies any freeblock tasks and performs associated processing and clean-up. The foreground scheduler selects a new foreground request, if any are pending, adds it to the dispatch queue, and invokes the background scheduler to possibly add freeblock requests. Then, *dadone()* proceeds normally.

Freeblock system call translator: The system call translator implements the application-level API. Doing

so consists of translating system calls to in-kernel calls and managing the flow of data between the freeblock subsystem and the user-level application. When a freeblock task completes, the translator's *callback fn* appends a record to the associated session's buffers and, if the buffers were empty, awakens any waiting application processes. When the freeblock subsystem reads data faster than the application processes it, the buffers associated with the session fill up and flow control is needed. When this happens, the translator uses the *fb_suspend* call, suspending subsequent freeblock requests for the tasks associated with the given session. When the application fetches records and thereby clears space, the translator uses *fb_resume* to re-enable the associated freeblock tasks. When an application exits or calls *sys_fb_close*, the translator clears all state maintained by the freeblock system on behalf of the application's session(s).

5.2 Background scheduler algorithms

The background scheduler includes algorithms for utilizing otherwise wasted rotational latency gaps and for detecting and using disk idle time.

Rotational latency usage: Recall that, during busy disk periods, rotational latency gaps can be filled with background media transfers. Our freeblock subsystem uses algorithms similar to those described by Lumb et al. [20, 21], modified to use less non-idle CPU time and to support fairness and priorities among freeblock sessions.

The search for suitable background transfers proceeds in two phases. The first phase checks only a few tracks for potential background transfers adding an insignificant amount of computation ($<<8\%$) to a busy CPU. The second phase only runs when the CPU is otherwise in the idle loop. It searches all other options to refine the best choice found until the request needs to be sent.

Prior algorithms greedily scheduled freeblock requests, assuming all were equal. As shown in Section 6.6, this can lead to poor behavior when freeblock sessions are mixed. In particular, full disk scans can starve other sessions. We introduce fairness, as well as support for priorities, using a simple form of lottery scheduling [30]. The initial tickets allocated to each session are proportional to its assigned priority.

The lottery determines both which pending tasks are considered, since there is limited CPU time for searching, and which viable option found is selected. During the first phase, which runs for a short *quanta* of time, as described in [29], cylinders closest to the source and destination cylinders with pending tasks from the winning session are considered. Any option from the winning session found will be selected. In addition, all pending tasks on the destination cylinder and within one cylinder of the source are considered; these are the most likely locations of viable options, reducing the odds that the rotational latency gap goes unused. During a second phase, all pending tasks from the winning session are considered and given strict priority over pending tasks from other sessions.

Idle time detection and usage: Previous research [13, 26] reports that most idle periods are a few milliseconds in length and that long idle time periods come in multi-second durations. Our freeblock subsystem utilizes both. Borrowing from prior work [13], a simple threshold (of 20ms) is used to identify likely idle periods. During short idle times, the scheduler considers pending freeblock reads on the same track. Such data can be read and cached in the device driver with minimal impact on foreground access patterns, because no mechanical delays are induced and no disk prefetching is lost.

For each quanta of a long idle period, a session is selected via the lottery. Pending tasks of the winning session are scheduled, starting with the most difficult to service using rotational latency gaps: those near the innermost and outermost cylinders.

Algorithm summary: Our outside-the-disk freeblock scheduler has the same "imperfect knowledge and control" limitations described by Lumb et al. [20], and thereby loses about 35% of the potential free bandwidth. An implementation embedded in a disk drive could be expected to provide correspondingly higher free bandwidth to applications. The introduction of conservative CPU usage further reduces free bandwidth utilization by 5–10%. Our evaluations show that the remaining free bandwidth is adequate for most background applications. Detailed description and evaluation of the freeblock subsystem's data structures and algorithms are available in [29].

6 Evaluation

This section evaluates how effectively the framework supports the three background applications.

6.1 Experimental setup

All experiments are run on a system with a dual 1GHz Pentium III, 384MB of main memory, an Intel 440BX chipset with a 33MHz, 32bit PCI bus, and an Adaptec AHA-2940 Ultra2Wide SCSI controller. Unless otherwise stated, the experiments use a Seagate Cheetah 36ES disk drive with a capacity of 18GB and results are averaged from at least five runs. Two implementations of the freeblock subsystem are used: one in the FreeBSD device driver and one in user-level Linux. The user-level Linux implementation can either do direct SCSI reads and writes or communicate with a simulated storage device implemented by DiskSim [4]. All implementations use the same scheduling core and conservatism factors

used in the FreeBSD implementation.

Three benchmarks are used throughout the evaluation section. The **synthetic benchmark** is a multi-threaded program that continuously issues small (4KB-8KB) read and write I/Os to disk, with a read-write ratio of 2:1, keeping two requests at the disk queue at all times.

The **TPC-C benchmark** [8] simulates an on-line transaction processing database workload, where each transaction consists of a few read-modify-write operations to a small number of records. We ran TPC-C on the Shore database storage manager [5]. We configured Shore and TPC-C to use 8KB pages, a 32MB page buffer pool, 50 warehouses (covering approximately 70% of the Seagate disk's capacity) and 10 clients per warehouse. The Shore volume is a file stored in FreeBSD's FFS file system. Thus, an I/O generated by Shore goes through the file system buffer cache. Performance of a TPC-C benchmark is measured in TPC-C transactions completed per minute (TpmC)

The **Postmark benchmark** [18] was designed to measure the performance of a file system used for electronic mail, netnews and web-based services. It creates a large number of small files and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 100,000 transactions on 800,000 files in 10,000 directories. File sizes range from 10KB to 20KB. The biases are Postmark's defaults: read/append=5, create/delete=5.

6.2 Freeblock subsystem effectiveness

This section briefly evaluates the freeblock subsystem's effectiveness. Figure 5 shows the efficiency of the freeblock subsystem, as a function of disk utilization, with the synthetic benchmark as the foreground application. A background disk scan registers a freeblock task to read every block of the disk. The *callback fn* re-registers each block as it is read, thus maintaining a constant number of blocks wanted. The synthetic benchmark is modified slightly so that the number of I/Os per second can be varied; the request inter-arrival times are exponentially distributed with uniform means.

The freeblock subsystem ensures that background applications make forward progress, irrespective of the disk's utilization. As expected, the progress is fastest when the disk is mostly idle. The amount of free bandwidth is lowest when the system is 40-60% utilized, because short idle times are less useful than either rotational latency gaps or long idle times. Regardless of utilization, foreground requests are affected by less than 2%. For a full evaluation of the freeblock infrastructure and algorithms, please refer to [29].

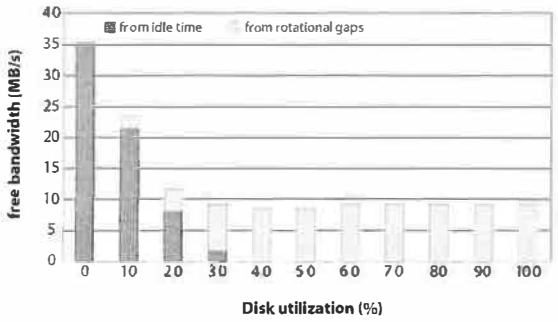
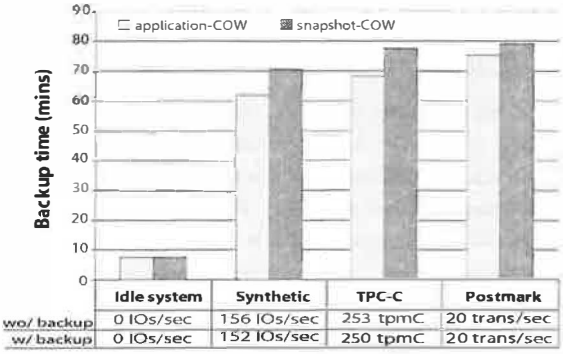


Figure 5: **Freeblock subsystem efficiency.** This diagram illustrates the instantaneous free bandwidth extracted for a background disk scan as a function of the disk's utilization. When the foreground workload is light, idle time is the main source of free bandwidth. When the foreground workload intensifies, the free bandwidth comes from rotational latency gaps.



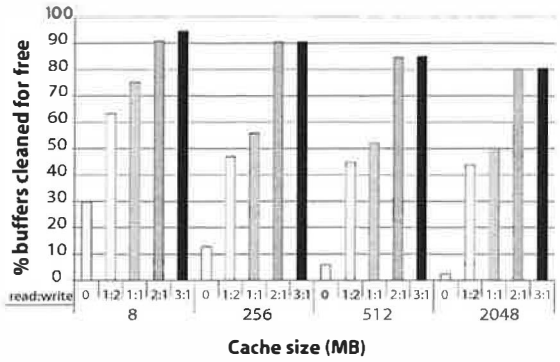
	Idle system	Synthetic	TPC-C	Postmark
wo/ backup	0 IOs/sec	156 IOs/sec	253 tpmC	20 trans/sec
w/ backup	0 IOs/sec	152 IOs/sec	250 tpmC	20 trans/sec

Figure 6: **Snapshot-based backup efficiency.** This diagram illustrates the efficiency of the backup application when backing up 70% of the Cheetah 36ES disk (18GB). The foreground workload is affected less than 2% during the background backup as a result of access time mispredictions that result from the outside-the-disk implementation of freeblock scheduling.

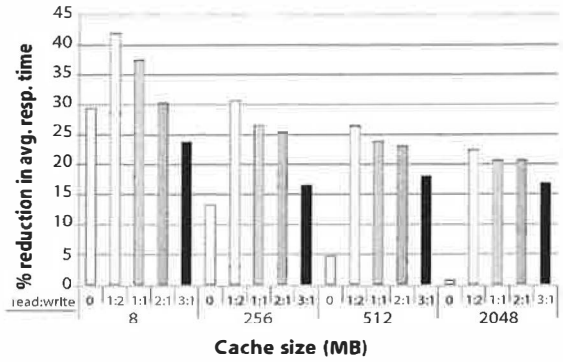
6.3 Snapshot-based backup

This section evaluates the backup application described in Section 4.1. We evaluate both the application-copy-on-write (application-COW) and snapshot-copy-on-write (snapshot-COW) strategies in the FreeBSD kernel. When application-COW is used, all subsequent modifications to a block that the snapshot claims are sent to a new location. When snapshot-COW is used, all subsequent modifications go to the original location of the block, and the snapshot system makes a private copy of the block for itself. The native snapshot implementation in FreeBSD supports only snapshot-COW; we instrumented the kernel so that we could evaluate application-COW as well.

Figure 6 shows the performance of our backup application when sharing the system with the three foreground benchmarks. The table beneath the graph shows that the impact of the concurrent backup on foreground



(a) **Cleaning efficiency (% cleaned for free)**



(b) **Response time reduction from free write-backs**

Figure 7: **Free cache cleaning with LRU replacement and syncer daemon.** These graphs illustrate the efficiency of freeblock scheduling and the impact it makes on the average response time of reads given a LRU replacement policy and a syncer daemon that guarantees no dirty block will stay dirty for longer than 30 seconds. The x-axis contains the cache size and the read-write ratio. A read-write ratio of 0 means that all requests are writes.

performance is less than 2%. During the synthetic benchmark, the backup is completed faster than during Postmark or TPC-C. This is so because the synthetic benchmark's requests are uniformly distributed around the disk, maximizing the scheduler's opportunities. The backup is slightly faster under TPC-C than under Postmark for several reasons. First, Postmark is single-threaded and has short disk idle periods, but that are too short to be exploited. Thus, fewer freeblock opportunities arise during any given time period. Second, the cache manager successfully coalesces many small dirty buffers for Postmark and thus issues larger I/Os to the device, which reduces the effectiveness of freeblock scheduling further.

In the idle time case, the streaming bandwidth is about 35MB,[6] and the backup completes in little over 8 minutes. The graph also shows that the application-COW results in more efficient use of free bandwidth. This is because, with snapshot-COW, the backup application wastes some bandwidth reading blocks that have been modified; it then needs to re-register reads for the new locations of those blocks. The overall effect however, is less then a 15% increase in the time to complete the backup. Thus, a backup application based on our framework can be effective with whichever implementation is used by a particular system.

6.4 Buffer cache cleaner

We evaluate the efficiency of the buffer cache cleaner designed using our framework with both controlled experiments (using the Linux user-level implementation with a simulated cache and direct SCSI reads and writes on the Seagate disk) and the implementation in FreeBSD. The controlled experiments are used to understand the relationship between the efficiency of the cache cleaner and the size of the cache, the workload presented, and the replacement and persistence policies. The metrics of interest are the percentage of dirty blocks cleaned for free and the reduction in average response time of other requests. In all buffer cache experiments, the idle-time detector does not detect enough idle time to be helpful.

The controlled experiments use a version of the synthetic benchmark. As indicated, we vary the read-write ratio and the simulated cache size while keeping the size of the requests the same (4KB-8KB).

Effect of cache size and read-write ratio: Figure 7 shows the efficiency of the cleaner and its impact on the overall response time as a function of the workload's read-write ratio and the cache size. The replacement policy is least-recently used (LRU), and the persistence policy guarantees that no dirty buffer will stay dirty for longer than 30 seconds. High and low water-marks are used to address space exhaustion: whenever the number of dirty buffers in the cache hits the high water-mark, the cache manager cleans up as many buffers as needed until the low water-mark is reached. Mimicking the notation used by FreeBSD's cache manager, a *syncer daemon* implements the persistence policy, and a *buffer daemon* implements the logic that checks the high and low water-marks.

Several observations can be made from Figure 7. First, as the read-write ratio increases, a larger percentage of the dirty buffers can be cleaned for free, because more and more freeblock opportunities are created. Writes do not go to disk immediately because of write-back caching. Instead, they go to disk as a re-

[6] The reported streaming bandwidth of the disk is 40MB/s. But, due to head switch delays when changing tracks, the observed streaming bandwidth is about 35MB/s.

sult of the syncer's work or buffer daemon's work. In both cases, they go to disk in large bursts. Hence, the foreground scheduler (using SPTF) does a good job in scheduling, reducing the freeblock scheduler's chances of finding rotational gaps to use.

Second, as the read-write ratio increases (beyond 1:2), the impact of free cleaning on the average response time decreases. This is a direct consequence of the decreasing number of writes (and, hence, dirty buffers) in the system. Third, the efficiency of the freeblock subsystem slightly decreases with increasing cache size. The reason is that every time the syncer or buffer daemons wake up, they have a larger number of dirty buffers to flush. Again, the foreground scheduler reduces the freeblock scheduler's chances of finding rotational gaps to use. However, we observed that the opposite happens, i.e. the efficiency of the scheduler *increases*, when no persistence policy is used.

Effect of replacement and persistence policies: Figure 8 examines the efficiency of the cache cleaner and its impact on the average response time under different replacement and persistence policies. The cache size is kept fixed (512MB) and the read-write ratio is 1:1. In addition to LRU, two other replacement policies are evaluated. The SPTF-Evict policy is similar to LRU, but instead of replacing dirty entries in an LRU fashion, the entries closest to the disk head position are replaced first. The FREE-CLEAN (FC) policy chooses to replace a *clean* entry that has been recently cleaned for free (if none exists, it reverts to LRU). By replacing a clean entry from the cache, FREE-CLEAN attempts to let the remaining dirty buffers stay a little longer in the system so that they may be written out for free.

All three replacement policies are evaluated with, and without, a syncer daemon. A syncer daemon places a hard limit (30 seconds in our case) on the time the freeblock subsystem has to clean any dirty buffers for free. Hence, fewer buffers are cleaned for free under this policy, irrespective of the replacement policy used. A cache comprised of non-volatile RAM, on the other hand, does not need such a persistence policy.

The SPTF-Evict policy reduces the effectiveness of the freeblock subsystem most, thereby reducing its benefit to the average response time. This is because no write task can be satisfied during write I/Os that happen as a result of the buffer daemon (because the dirty buffer closest to the disk head is written first, there are no other dirty buffers freeblock scheduling can squeeze in between foreground requests). Write tasks can still be satisfied during write I/Os that happen because of the syncer daemon. In the case when no syncer daemon is used, all writes happen due to the buffer daemon, hence dirty buffers can be cleaned for free only during foreground read requests.

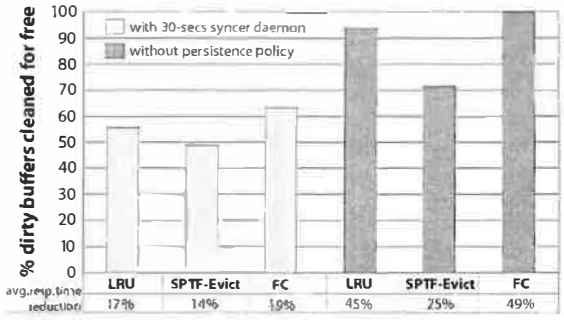


Figure 8: **Comparison of replacement and persistence policies.** These graphs illustrate the efficiency of the cache cleaner on a system under different replacement and persistence policies

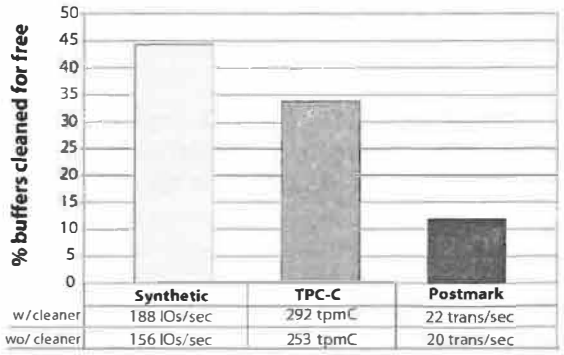


Figure 9: **Cache cleaner efficiency in FreeBSD.** The throughput metrics below each bar show overall performance with and without free write-backs.

FC is the best policy as far as the cache cleaner is concerned. By leaving the dirty buffers in the cache a little longer, it can clean more of them for free. But, there could be a detrimental effect on cache hit rate, and the cache cleaning benefit observed is quite small.

Cache cleaning in FreeBSD: Figure 9 illustrates the efficiency of the real cache cleaner, implemented as part of FreeBSD's cache manager. At most, 3/4 of the system's 384MB of RAM are devoted to the I/O buffering subsystem. The read-write ratio of the synthetic benchmark is 1:1, the observed read-write ratio of TPC-C is approximately 1:1, and the observed read-write ratio of Postmark is approximately 1:3. In all three cases, a sizeable percentage of the dirty buffers are cleaned for free. Postmark benefits less than the other benchmarks for the same reasons it lagged in the backup evaluation: write-back clustering and unusable short idle periods.

6.5 Layout reorganizer

To evaluate the effectiveness of our reorganizer, we performed a variety of controlled experiments. The foreground workload is the synthetic benchmark, which keeps the disk 100% utilized. To avoid corruption of the

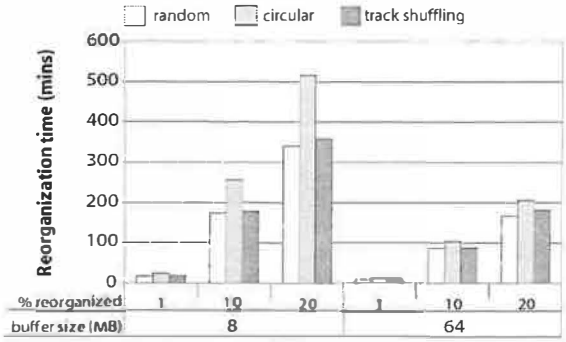


Figure 10: **Layout reorganizer efficiency.** Each bar cluster shows the time required for the three reorganization actions for a different staging buffer capacity the percentage of the disk space being reorganized.

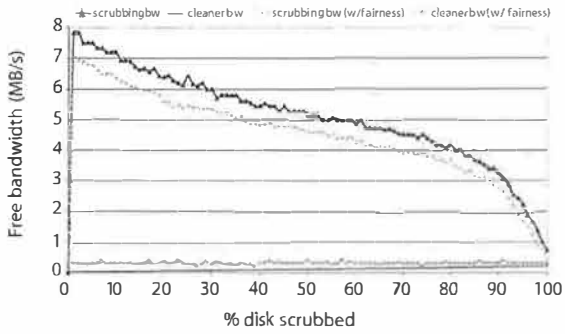


Figure 11: **Disk scrubbing and cache cleaning.** This figure shows two concurrent background applications, disk scrubbing and cache cleaning, in a system with and without fairness.

FFS file system in FreeBSD, the experiments are run in the user-level Linux environment. In all experiments, the base unit the reorganizer is interested in moving at any time is 8KB (specified by the *blksize* parameter). Three different reorganization actions are explored.

Random reorganization: Random 8KB blocks on the disk are moved to other random locations. Few blocks have dependencies using this method.

Circular random reorganization: A list of unique random 8KB blocks is created, and each block is moved to the location of the next block in the list. This creates the longest dependency chain possible: one including every block to be reorganized.

Track shuffling: Similar to the random block reorganization action above, but whole tracks are shuffled instead of blocks.

We evaluated each action reorganizing from 1% to 20% of the disk. Research on reorganization techniques indicates that this range is generally the most effective amount of the disk to reorganize [1, 16]. The results are shown in Figure 10. Tests with more dependencies, like circular, take longer than those with few dependencies. They also benefit more from an increase in buffer size.

The results are encouraging, showing that up to 20% of the disk can be reorganized in a few hours on a fully busy disk.

6.6 Application fairness and priorities

This section briefly evaluates the fairness of the scheduling algorithms. Two applications compete for the free bandwidth: a simple disk scrubber and the cache cleaner evaluated above. The disk scrubber simply tries to read all blocks of the disk once, without worrying about consistency issues (hence it doesn't use the snapshot system). The experiment is run until the disk scrubber has read all blocks of the 18GB Seagate disk.

The bandwidths dedicated to the scrubber and cache cleaner applications are measured. In the original case, the freeblock scheduler's fairness mechanisms are disabled and the scheduling algorithms lean toward a greedy approach. In the fair system, lottery scheduling makes sure that both applications are treated fairly. Both applications are assigned the default priority. The cache size is fixed at 512MB, the replacement policy is LRU, and the persistence policy is implemented using the 30-sec syncer daemon. The read-write ratio of the foreground workload is 1:1.

Figure 11 shows the distribution of bandwidth with and without fairness. The bandwidth given to the cache cleaner increases from almost nothing to about 0.3MB/s when priorities are used. This bandwidth is very close to 0.34MB/s, which is the bandwidth the cache cleaner would get if it were the only background application in the system. The bandwidth of the scrubber, on the other hand, falls by a little more than the gained bandwidth of the cache cleaner. This 2-5% loss in efficiency can be attributed to the scheduler's decision to treat the cache cleaner in a fair manner, thereby spending an equal time searching for opportunities that satisfy tasks of that application. These opportunities are smaller when compared to the opportunities of the scrubber.

7 Summary

This paper describes a programming framework for developing background disk maintenance applications. With several case studies, we show that such applications can be adapted to this framework effectively. A freeblock subsystem can provide disk access to these applications, using freeblock scheduling and idle time, with minimal impact on the foreground workload.

Acknowledgements

We thank Vinod Das Krishnan, Steve Muckle and Brian Railing for assisting with porting of freeblock scheduling code into FreeBSD. Special thanks to Chet Juszczak (our shepherd) and to all anonymous reviewers who

provided much constructive feedback. We also thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, LSI Logic, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. This work is funded in part by NSF grants CCR-0113660 and CCR-0205544.

References

[1] S. Akyurek and K. Salem. *Adaptive block rearrangement.* CS-TR-2854. University of Maryland, February 1992.

[2] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: making the fast case common. Workshop on Input/Output in Parallel and Distributed Systems, pages 10–22. ACM Press, 1999.

[3] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. IEEE International Conference on Multimedia Computing and Systems, pages 400–405. IEEE, 1999.

[4] J. S. Bucy and G. R. Ganger. *The DiskSim simulation environment version 3.0 reference manual.* Technical Report CMU-CS-03-102. Department of Computer Science Carnegie-Mellon University, Pittsburgh, PA, January 2003.

[5] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. ACM SIGMOD International Conference on Management of Data. Published as *SIGMOD Record*, **23**(2):383–394, 1994.

[6] S. C. Carson and S. Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering*, **18**(1):44–54, January 1992.

[7] A. L. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: a survey of backup techniques. Joint NASA and IEEE Mass Storage Conference, 1998.

[8] T. P. P. Council. TPC Benchmark C. Number Revision 5.1.0, 2002.

[9] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. ACM Symposium on Operating System Principles. Published as *Operating System Review*, **33**(5):247–260, December 1999.

[10] B. Duhl, S. Halladay, and P. Mankekar. Disk backup performance considerations. International Conference on Management and Performance Evaluation of Computer Systems, pages 646–663, 1984.

[11] R. M. English and A. A. Stepanov. Loge: a self-organizing disk controller. Winter USENIX Technical Conference, pages 237–251. Usenix, 20–24 January 1992.

[12] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, **18**(2):127–153. ACM Press, May 2000.

[13] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. Winter USENIX Technical Conference, pages 201–212. USENIX Association, 1995.

[14] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. Winter USENIX Technical Conference, pages 235–246. USENIX Association, 1994.

[15] R. Y. Hou, J. Menon, and Y. N. Patt. Balancing I/O response time and disk rebuild time in a RAID5 disk array. Hawaii International Conference on Systems Sciences, January 1993.

[16] W. Hsu. *Dynamic Locality Improvement Techniques for Increasing Effective Storage Performance.* PhD thesis, published as UCB/CSD-03-1223. University of California at Berkeley, January 2003.

[17] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. Symposium on Operating Systems Design and Implementation, pages 239–249. ACM, Winter 1998.

[18] J. Katcher. *PostMark: a new file system benchmark.* Technical report TR3022. Network Appliance, October 1997.

[19] D. Kotz. Disk-directed I/O for MIMD multiprocessors. Symposium on Operating Systems Design and Implementation, pages 61–74. USENIX Association, 14–17 November 1994.

[20] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. Conference on File and Storage Technologies, pages 275–288. USENIX Association, 2002.

[21] C. R. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle, and E. Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. Symposium on Operating Systems Design and Implementation, pages 87–102. USENIX Association, 2000.

[22] M. K. McKusick. Running 'fsck' in the background. BSDCon Conference, 2002.

[23] A. Molano, K. Juvva, and R. Rajkumar. Real-time filesystems. Guaranteeing timing constraints for disk accesses in RT-Mach. Proceedings Real-Time Systems Symposium, pages 155–165. IEEE Comp. Soc., 1997.

[24] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. Symposium on Operating Systems Design and Implementation. Published as *Operating System Review*, pages 15–28. ACM, 1998.

[25] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: file system based asynchronous mirroring for disaster recovery. Conference on File and Storage Technologies, pages 117–129. USENIX Association, 2002.

[26] C. Ruemmler and J. Wilkes. UNIX disk access patterns. Winter USENIX Technical Conference, pages 405–420, 1993.

[27] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. Published as *Performance Evaluation Review*, **26**(1):44–55, 1998.

[28] D. C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggreggate file I/O latency. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **31**(5):252–263. ACM, 1997.

[29] E. Thereska, J. Schindler, C. R. Lumb, J. Bucy, B. Salmon, and G. R. Ganger. *Design and implementation of a freeblock subsystem.* Technical report CMU-PDL-03-107. December 2003.

[30] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. Symposium on Operating Systems Design and Implementation, pages 1–11. Usenix Association, 14–17 November 1994.

[31] R. Y. Wang, D. A. Patterson, and T. E. Anderson. Virtual log based file systems for a programmable disk. Symposium on Operating Systems Design and Implementation, pages 29–43. ACM, 1999.

[32] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. Symposium on Operating Systems Design and Implementation, pages 243–258. USENIX Association, 2000.

[33] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and scheduling an eager-writing disk array for a transaction processing workload. Conference on File and Storage Technologies, pages 289–304. USENIX Association, 2002.

Integrating Portable and Distributed Storage

Niraj Tolia[†‡], Jan Harkes[†], Michael Kozuch[‡], and M. Satyanarayanan[†‡]
[†]Carnegie Mellon University and [‡]Intel Research Pittsburgh

Abstract

We describe a technique called *lookaside caching* that combines the strengths of distributed file systems and portable storage devices, while negating their weaknesses. In spite of its simplicity, this technique proves to be powerful and versatile. By unifying distributed storage and portable storage into a single abstraction, lookaside caching allows users to treat devices they carry as merely performance and availability assists for distant file servers. Careless use of portable storage has no catastrophic consequences. Experimental results show that significant performance improvements are possible even in the presence of stale data on the portable device.

1 Introduction

Floppy disks were the sole means of sharing data across users and computers in the early days of personal computing. Although they were trivial to use, considerable discipline and foresight was required of users to ensure data consistency and availability, and to avoid data loss — if you did not have the right floppy at the right place and time, you were in trouble! These limitations were overcome by the emergence of distributed file systems such as NFS [24], Netware [8], LanManager [34], and AFS [7]. In such a system, responsibility for data management is delegated to the distributed file system and its operational staff.

Personal storage has come full circle in the recent past. There has been explosive growth in the availability of USB- and Firewire-connected storage devices such as flash memory keychains and portable disk drives. Although very different from floppy disks in capacity, data transfer rate, form factor, and longevity, their usage model is no different. In other words, they are just glorified floppy disks and suffer from the same limitations mentioned above. Why then are portable storage devices in such demand today? Is there a way to use them that avoids the messy mistakes of the past, where a user was often awash in floppy disks trying to figure out which one had the latest version of a specific file? If loss, theft or destruction of a portable storage device occurs, how can one prevent catastrophic data loss? Since human attention grows ever more scarce, can we reduce the data management demands on attention and discipline in the use of portable devices?

We focus on these and related questions in this paper. We describe a technique called *lookaside caching* that combines the strengths of distributed file systems and portable storage devices, while negating their weaknesses. In spite of its simplicity, this technique proves to be powerful and versatile. By unifying "storage in the cloud" (distributed storage) and "storage in the hand" (portable storage) into a single abstraction, lookaside caching allows users to treat devices they carry as merely performance and availability assists for distant file servers. Careless use of portable storage has no catastrophic consequences.

We begin in Section 2 by examining the strengths and weaknesses of portable storage and distributed file systems. We describe the design of lookaside caching in Section 3 followed by a discussion of related work in Section 4. Section 5 describes the implementation of lookaside caching. We quantify the performance benefit of lookaside caching in Section 6, using three different benchmarks. We explore broader use of lookaside caching in Section 7, and conclude in Section 8 with a summary.

2 Background

To understand the continuing popularity of portable storage, it is useful to review the strengths and weaknesses of portable storage and distributed file systems. While there is considerable variation in the designs of distributed file systems, there is also a substantial degree of commonality across them. Our discussion below focuses on these common themes.

Performance: A portable storage device offers uniform performance at all locations, independent of factors such as network connectivity, initial cache state, and temporal locality of references. Except for a few devices such as floppy disks, the access times and bandwidths of portable devices are comparable to those of local disks. In contrast, the performance of a distributed file system is highly variable. With a warm client cache and good locality, performance can match local storage. With a cold cache, poor connectivity and low locality, performance can be intolerably slow.

Availability: If you have a portable storage device in hand, you can access its data. Short of device failure, which is very rare, no other common failures prevent data access. In contrast, distributed file systems are susceptible to network failure, server failure, and a wide range of operator errors.

Robustness: A portable storage device can easily be lost, stolen or damaged. Data on the device becomes permanently inaccessible after such an event. In contrast, data in

a distributed file system continues to be accessible even if a particular client that uses it is lost, stolen or damaged. For added robustness, the operational staff of a distributed file system perform regular backups and typically keep some of the backups off site to allow recovery after catastrophic site failure. Backups also help recovery from user error: if a user accidentally deletes a critical file, he can recover a backed-up version of it. In principle, a highly disciplined user could implement a careful regimen of backup of portable storage to improve robustness. In practice, few users are sufficiently disciplined and well-organized to do this. It is much simpler for professional staff to regularly back up a few file servers, thus benefiting all users.

Sharing/Collaboration: The existence of a common name space simplifies sharing of data and collaboration between the users of a distributed file system. This is much harder if done by physical transfers of devices. If one is restricted to sharing through physical devices, a system such as PersonalRAID [26] can be valuable in managing complexity.

Consistency: Without explicit user effort, a distributed file system presents the latest version of a file when it is accessed. In contrast, a portable device has to be explicitly kept up to date. When multiple users can update a file, it is easy to get into situations where a portable device has stale data without its owner being aware of this fact.

Capacity: Any portable storage device has finite capacity. In contrast, the client of a distributed file system can access virtually unlimited amounts of data spread across multiple file servers. Since local storage on the client is merely a cache of server data, its size only limits working set size rather than total data size.

Security: The privacy and integrity of data on portable storage devices relies primarily on physical security. A further level of safety can be provided by encrypting the data on the device, and by requiring a password to mount it. These can be valuable as a second layer of defense in case physical security fails. Denial of service is impossible if a user has a portable storage device in hand. In contrast, the security of data in a distributed file system is based on more fragile assumptions. Denial of service may be possible through network attacks. Privacy depends on encryption of network traffic. Fine-grain protection of data through mechanisms such as access control lists is possible, but relies on secure authentication using a mechanism such as Kerberos [28].

Ubiquity: A distributed file system requires operating system support. In addition, it may require environmental support such as Kerberos authentication and specific firewall configuration. Unless a user is at a client that meets all of these requirements, he cannot access his data in a distributed file system. In contrast, portable storage only depends on widely-supported low-level hardware and software interfaces. If a user sits down at a random machine, he can be much more confident of accessing data from portable storage in his possession than from a remote file server.

3 Lookaside Caching

Our goal is to exploit the performance and availability advantages of portable storage to improve these same attributes in a distributed file system. The resulting design should preserve all other characteristics of the underlying distributed file system. In particular, there should be no compromise of robustness, consistency or security. There should also be no added complexity in sharing and collaboration. Finally, the design should be tolerant of human error: improper use of the portable storage device (such as using the wrong device or forgetting to copy the latest version of a file to it) should not hurt correctness.

Lookaside caching is an extension of AFS2-style whole-file caching [7] that meets the above goals. It is based on the observation that virtually all distributed file system protocols provide separate remote procedure calls (RPCs) for access of meta-data and access of data content. Lookaside caching extends the definition of meta-data to include a cryptographic hash of data content. This extension only increases the size of meta-data by a modest amount: just 20 bytes if SHA-1 [15] is used as the hash. Since hash size does not depend on file length, it costs very little to obtain and cache hash information even for many large files. Using POSIX terminology, caching the results of "`ls -lR`" of a large tree is feasible on a small client, even if there is not enough cache space for the contents of all the files in the tree. This continues to be true even if one augments `stat` information for each file or directory in the tree with its SHA-1 hash.

Once a client possesses valid meta-data for an object, it can use the hash to redirect the fetch of data content. If a mounted portable storage device has a file with matching length and hash, the client can obtain the contents of the file from the device rather than from the file server. Whether it is beneficial to do this depends on factors such as file size, network bandwidth, and device transfer rate. The important point is that possession of the hash gives a degree of freedom that clients of a distributed file system do not possess today.

Since lookaside caching treats the hash as part of the meta-data, there is no compromise in consistency. The underlying cache coherence protocol of the distributed file system determines how closely client state tracks server state. There is no degradation in the accuracy of this tracking if the hash is used to redirect access of data content. To ensure no compromise in security, the file server should return a null hash for any object on which the client only has permission to read the meta-data.

Lookaside caching can be viewed as a degenerate case of the use of *file recipes,* as described by Tolia *et al.* [31]. In that work, a recipe is an XML description of file content that enables block-level reassembly of the file from content-addressable storage. One can view the hash of a file as the smallest possible recipe for it. The implementation using recipes is considerably more complex than our support for lookaside caching. In return for this complexity, synthesis

from recipes may succeed in many situations where lookaside fails.

4 Related Work

Lookaside caching has very different goals and design philosophy from systems such as PersonalRAID [26], Segank [25], and Footloose [18]. Our starting point is the well-entrenched base of distributed file systems in existence today. We assume that these are successful because they offer genuine value to their users. Hence, our goal is to integrate portable storage devices into such a system in a manner that is minimally disruptive of its existing usage model. In addition, we make no changes to the native file system format of a portable storage device; all we require is that the device be mountable as a local file system at any client of the distributed file system. In contrast, all the above systems takes a much richer view of the role of portable storage devices. They view them as first-class citizens rather than as adjuncts to a distributed file system. They also use customized storage layouts on the devices. Therefore, our design and implementation are much simpler, but also more limited in functionality.

Another project with overlapping goals is the Personal Server [32] effort. This system tries to integrate computation, communication, and storage to provide ubiquitous access to personal information and applications. However, being a mobile computer, it is more heavyweight in terms of the hardware requirements. There are also a number of commercial solutions providing mobility solutions through the use of portable storage devices. Migo [12], one of these products, has combined a USB portable storage device with synchronization software for personal files, email, and other settings. However, these solutions focus exclusively on the use of the portable device and do not integrate network storage.

The use of cryptographic hashes to describe data has been explored earlier in a variety of different contexts. Spring *et al.* [27] used the technique to identify and remove redundant network traffic. The Single Instance Storage [3] and the Venti [20] systems use cryptographic hashes to remove duplicate content at the file and block level respectively. Unlike lookaside caching, a number of other systems such as CASPER [31] and LBFS [14] prefer to further subdivide objects. This slightly more complicated approach usually uses an algorithm similar to the Rabin fingerprinting technique [10, 21]. For lookaside caching, it was a conscious decision to favor the simplest possible design. It is also well known that the use of hashes can leak information. In the context of lookaside caching, fetching a SHA-1 hash without fetching the corresponding contents can indicate that the client already possessed the data. As shown by Mogul *et al.* [13], this can allow a malicious server to inspect a client's cache. The most obvious solution is to only allow lookaside caching with trusted servers. As we believe that the predominant use of lookaside caching will be with trusted servers, this solution should not significantly impact users.

5 Prototype Implementation

We have implemented lookaside caching in the Coda file system on Linux. The user-level implementation of Coda client cache manager and server code greatly simplified our effort since no kernel changes were needed. The implementation consists of four parts: a small change to the client-server protocol; a quick index check (the "lookaside") in the code path for handling a cache miss; a tool for generating lookaside indexes; and a set of user commands to include or exclude specific lookaside devices.

The protocol change replaces two RPCs, `ViceGetAttr()` and `ViceValidateAttrs()` with the extended calls `ViceGetAttrPlusSHA()` and `ViceValidateAttrsPlusSHA()` that have an extra parameter for the SHA-1 hash of the file. `ViceGetAttr()` is used to obtain meta-data for a file or directory, while `ViceValidateAttrs()` is used to revalidate cached meta-data for a collection of files or directories when connectivity is restored to a server. Our implementation preserves compatibility with legacy servers. If a client connects to a server that has not been upgraded to support lookaside caching, it falls back to using the original RPCs mentioned above.

The lookaside occurs just before the execution of the `ViceFetch()` RPC to fetch file contents. Before network communication is attempted, the client consults one or more lookaside indexes to see if a local file with identical SHA-1 value exists. Trusting in the collision resistance of SHA-1 [11], a copy operation on the local file can then be a substitute for the RPC. To detect version skew between the local file and its index, the SHA-1 hash of the local file is recomputed. In case of a mismatch, the local file substitution is suppressed and the cache miss is serviced by contacting the file server. Coda's consistency model is not compromised, although some small amount amount of work is wasted on the lookaside path.

The index generation tool walks the file tree rooted at a specified pathname. It computes the SHA-1 hash of each file and enters the filename-hash pair into the index file, which is similar to a Berkeley DB database [17]. The tool is flexible regarding the location of the tree being indexed: it can be local, on a mounted storage device, or even on a nearby NFS or Samba server. For a removable device such as a USB storage keychain or a DVD, the index is typically located right on the device. This yields a self-describing storage device that can be used anywhere. Note that an index captures the values in a tree at one point in time. No attempt is made to track updates made to the tree after the index is created. The tool must be re-run to reflect those updates. Thus, a lookaside index is best viewed as a collection of *hints* [30].

`cfs lka --clear`	*exclude all indexes*
`cfs lka +db1`	*include index db1*
`cfs lka -db1`	*exclude index db1*
`cfs lka --list`	*print lookaside statistics*

Figure 1. Lookaside Commands on Client

Dynamic inclusion or exclusion of lookaside devices is done through user-level commands. Figure 1 lists the relevant commands on a client. Note that multiple lookaside devices can be in use at the same time. The devices are searched in order of inclusion.

As mentioned earlier, the fact that our system does not modify the portable device's storage layout allows it to use any device that exports a generic file system interface. This allows files to be stored on the device in any manner chosen by the user, including the same tree structure as the distributed file system. For example, in the Kernel Compile benchmark described in Section 6.1, the portable device was populated by simply unarchiving a normal kernel source tree. The advantage of this is that user can still have access to the files in the absence of a network or even a distributed file system client. However, this also allows the user to edit files without the knowledge of the lookaside caching system. While recomputation of the file's hash at the time of use can expose the update, it is up to the user to manually copy the changes back into the distributed file system.

6 Evaluation

How much of a performance win can lookaside caching provide? The answer clearly depends on the workload, on network quality, and on the overlap between data on the lookaside device and data accessed from the distributed file system. To obtain a quantitative understanding of this relationship, we have conducted controlled experiments using three different benchmarks: a kernel compile benchmark, a virtual machine migration benchmark, and single-user trace replay benchmark. The rest of this section presents our benchmarks, experimental setups, and results.

6.1 Kernel Compile

6.1.1 Benchmark Description

Our first benchmark models a nomadic software developer who does work at different locations such as his home, his office, and a satellite office. Network connection quality to his file server may vary across these locations. The developer carries a version of his source code on a lookaside device. This version may have some stale files because of server updates by other members of the development team.

We use version 2.4 of the Linux kernel as the source tree in our benchmark. Figure 2 shows the measured degree of commonality across five different minor versions of the 2.4 kernel, obtained from the FTP site `ftp.kernel.org`. This

Kernel Version	Size (MB)	Files Same	Bytes Same	Release Date	Days Stale
2.4.18	118.0	100%	100%	02/25/02	0
2.4.17	116.2	90%	79%	12/21/01	66
2.4.13	112.6	74%	52%	10/23/01	125
2.4.9	108.0	53%	30%	08/16/01	193
2.4.0	95.3	28%	13%	01/04/01	417

This table shows key characteristics of the Linux kernel versions used in our compilation benchmark. In our experiments, the kernel being compiled was always version 2.4.18. The kernel on the lookaside device varied across the versions listed above. The second column gives the size of the source tree of a version. The third column shows what fraction of the files in that version remain the same in version 2.4.18. The number of bytes in those files, relative to total release size, is given in the fourth column. The last column gives the difference between the release date of a version and the release date of version 2.4.18.

Figure 2. Linux Kernel Source Trees

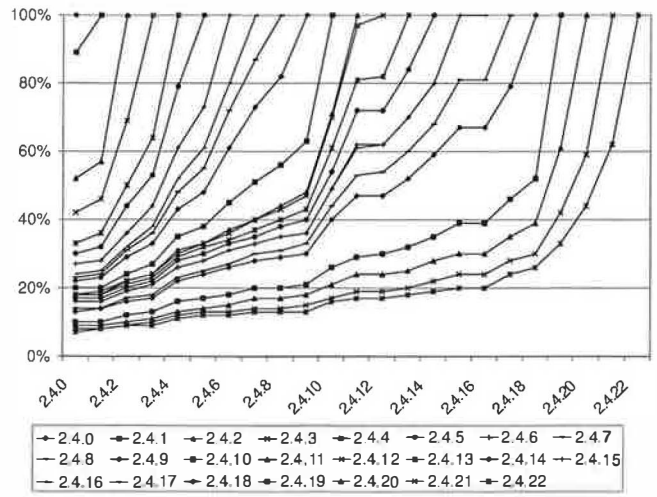
data shows that there is a substantial degree of commonality even across releases that are many weeks apart. Our experiments only use five versions of Linux, but Figure 3 confirms that commonality across minor versions exists for all of Linux 2.4. Although we do not show the corresponding figure, we have also confirmed the existence of substantial commonality across Linux 2.2 versions.

6.1.2 Experimental Setup

Figure 4 shows the experimental setup we used for our evaluation. The client contained a 3.0 GHz Pentium® 4 processor (with Hyper-Threading) with 2 GB of SDRAM. The file server contained a 2.0 GHz Pentium® 4 processor (without Hyper-Threading) with 1 GB of SDRAM. Both machines ran Red Hat 9.0 Linux and Coda 6.0.2, and were connected by 100 Mb/s Ethernet. The client file cache size was large enough to prevent eviction during the experiments, and the client was operated in write-disconnected mode. We ensured that the client file cache was always cold at the start of an experiment. To discount the effect of a cold I/O buffer cache on the server, a warming run was done prior to each set of experiments.

All experiments were run at four different bandwidth settings: 100 Mb/s, 10 Mb/s, 1 Mb/s and 100 Kb/s. We used a NISTNet network router [16] to control bandwidth. The router is simply a standard PC with two network interfaces running Red Hat 7.2 Linux and release 2.0.12 of the NISTNet software. No extra latency was added at 100 Mb/s and 10 Mb/s. For 1 Mb/s and 100 Kb/s, we configured NISTNet to add round trip latencies of 10 ms and 100 ms respectively.

The lookaside device used in our experiments was a 512 MB Hi-Speed USB flash memory keychain. The manufacturer of this device quotes a nominal read bandwidth of 48 Mb/s, and a nominal write bandwidth of 36 Mb/s. We conducted a set of tests on our client to verify these figures. Figure 6 presents our results. For all file sizes ranging from



Each curve above corresponds to one minor version of the Linux 2.4 kernel. That curve represents the measured commonality between the minor version and all previous minor versions. The horizontal axis shows the set of possible minor versions. The vertical axis shows the percentage of data in common. The rightmost point on each curve corresponds to 100% because each minor version overlaps 100% with itself.

Figure 3. Commonality Across Linux 2.4 Versions

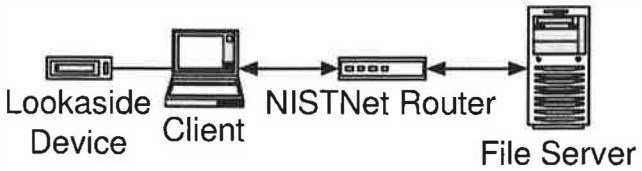


Figure 4. Experimental Setup

4 KB to 100 MB, the measured read and write bandwidths were much lower than the manufacturer's figures.

6.1.3 Results

The performance metric in this benchmark is the elapsed time to compile the 2.4.18 kernel. This directly corresponds to the performance perceived by our hypothetical software developer. Although the kernel being compiled was always version 2.4.18 in our experiments, we varied the contents of the portable storage device to explore the effects of using stale lookaside data. The portable storage device was unmounted between each experiment run to discount the effect of the buffer cache.

Figure 5 presents our results. For each portable device state shown in that figure, the corresponding "Files Same" and "Bytes Same" columns of Figure 2 bound the usefulness of lookaside caching. The "Days Stale" column indicates the staleness of device state relative to the kernel being compiled.

At the lowest bandwidth (100 Kb/s), the win due to lookaside caching is impressive: over 90% with an up-to-

File Size	Measured Data Rate	
	Read (Mb/s)	Write (Mb/s)
4 KB	6.3	7.4
16 KB	6.3	12.5
64 KB	16.7	25.0
256 KB	25.0	22.2
1 MB	28.6	25.8
10 MB	29.3	26.4
100 MB	29.4	26.5

This tables displays the measured read and write bandwidths for different file sizes on the portable storage device used in our experiments. To discount caching effects, we unmounted and remounted the device before each trial. For the same reason, all writes were performed in synchronous mode. Every data point is the mean of three trials; the standard deviation observed was negligible.

Figure 6. Portable Storage Device Performance

date device (improving from 9348.8 seconds to 884.9 seconds), and a non-trivial 10.6% (from 9348.8 seconds to 8356.7 seconds) with data that is over a year old (version 2.4.0)! Data that is over two months old (version 2.4.17) is still able to give a win of 67.8% (from 9348.8 seconds to 3011.2 seconds).

At a bandwidth of 1 Mb/s, the wins are still impressive. They range from 63% (from 1148.3 seconds to 424.8 seconds) with an up-to-date portable device, down to 4.7% (1148.3 seconds to 1094.3 seconds) with the oldest device state. A device that is stale by one version (2.4.17) still gives a win of 52.7% (1148.3 seconds to 543.6 seconds).

On a slow LAN (10 Mb/s), lookaside caching continues to give a strong win if the portable device has current data: 27.1% (388.4 seconds to 282.9 seconds). The win drops to 6.1% (388.4 seconds to 364.8 seconds) when the portable device is one version old (2.4.17). When the version is older than 2.4.17, the cost of failed lookasides exceeds the benefits of successful ones. This yields an overall loss rather than a win (represented as a negative win in Figure 5). The worst loss at 10 Mb/s is 8.4% (388.4 seconds to 421.1 seconds).

Only on a fast LAN (100 Mb/s) does the overhead of lookaside caching exceed its benefit for all device states. The loss ranges from a trivial 1.7% (287.7 seconds to 292.7 seconds) with current device state to a substantial 24.5% (287.7 seconds to 358.1 seconds) with the oldest device state. Since the client cache manager already monitors bandwidth to servers, it would be simple to suppress lookaside at high bandwidths. Although we have not yet implemented this simple change, we are confident that it can result in a system that almost never loses due to lookaside caching.

6.2 Internet Suspend/Resume

6.2.1 Benchmark Description

Our second benchmark is based on the application that forced us to rethink the relationship between portable storage and distributed file systems. *Internet Suspend/Resume*

Bandwidth	Lookaside Device State					
	No Device	2.4.18	2.4.17	2.4.13	2.4.9	2.4.0
100 Mb/s	287.7 (5.6)	292.7 (6.4) [-1.7%]	324.7 (16.4) [-12.9%]	346.4 (6.9) [-20.4%]	362.7 (3.4) [-26.1%]	358.1 (7.7) [-24.5%]
10 Mb/s	388.4 (12.9)	282.9 (8.3) [27.1%]	364.8 (12.4) [6.1%]	402.7 (2.3) [-3.7%]	410.9 (2.1) [-5.8%]	421.1 (12.8) [-8.4%]
1 Mb/s	1148.3 (6.9)	424.8 (3.1) [63.0%]	543.6 (11.5) [52.7%]	835.8 (3.7) [27.2%]	1012.4 (12.0) [11.8%]	1094.3 (5.4) [4.7%]
100 Kb/s	9348.8 (84.3)	884.9 (12.0) [90.5%]	3011.2 (167.6) [67.8%]	5824.0 (221.6) [37.7%]	7616.0 (130.0) [18.5%]	8356.7 (226.9) [10.6%]

These results show the time (in seconds) taken to compile the Linux 2.4.18 kernel. The column labeled "No Device" shows the time taken for the compile when no portable device was present and all data had to be fetched over the network. The column labeled "2.4.18" shows the results when all of the required data was present on the storage device and only meta-data (i.e. `stat` information) was fetched across the network. The rest of the columns show the cases where the lookaside device had versions of the Linux kernel older than 2.4.18. Each data point is the mean of three trials; standard deviations are in parentheses. The numbers in square brackets give the "win" for each case: that is, the percentage improvement over the "no device" case.

Figure 5. Time for Compiling Linux Kernel 2.4.18

(ISR) is a thick-client mechanism that allows a user to suspend work on one machine, travel to another location, and resume work on another machine there [9]. The user-visible state at resume is exactly what it was at suspend. ISR is implemented by layering a virtual machine (VM) on a distributed file system. The ISR prototype layers VMware on Coda, and represents VM state as a tree of 256 KB files.

A key ISR challenge is large VM state, typically many tens of GB. When a user resumes on a machine with a cold file cache, misses on the 256 KB files can result in significant performance degradation. This overhead can be substantial at resume sites with poor connectivity to the file server that holds VM state. If a user is willing to carry a portable storage device with him, part of the VM state can be copied to the device at suspend. Lookaside caching can then reduce the performance overhead of cache misses at the resume site. It might not always be possible to carry the entire VM state as writing it to the portable device may take too long for a user in a hurry to leave. In contrast, propagating updates to a file server can continue after the user leaves.

A different use of lookaside caching for ISR is based on the observation that there is often substantial commonality in VM state across users. For example, the installed code for applications such as Microsoft Office is likely to be the same for all users running the identical software release of those applications [3]. Since this code does not change until a software upgrade (typically many months apart), it would be simple to distribute copies of the relevant 256 KB files on DVD or CD-ROM media at likely resume sites.

Notice that lookaside caching is tolerant of human error in both of the above contexts. If the user inserts the wrong USB storage keychain into his machine at resume, stale data on it will be ignored. Similarly, use of the wrong DVD or CD-ROM does not hurt correctness. In both cases, the user sees slower performance but is otherwise unaffected.

Since ISR is intended for interactive workloads typical of laptop environments, we created a benchmark called the *Common Desktop Application (CDA)* that models an interactive Windows user. CDA uses Visual Basic scripting to drive Microsoft Office applications such as Word, Excel, Powerpoint, Access, and Internet Explorer. It consists of a total of 113 independently-timed operations such as `find-and-replace`, `open-document`, and `save-as-html`. Note that each of these macro-operations may result in more than one file system call within the VM and, consequently, multiple requests to Coda. Minor user-level actions such as keystrokes, object selection, or mouse-clicks are not timed.

6.2.2 Experimental Setup

Our experimental infrastructure consists of clients with 2.0 GHz Pentium® 4 processors connected to a server with a 1.2 GHz Pentium® III Xeon™ processor through 100 Mb/s Ethernet. All machines have 1 GB of RAM, and run Red Hat 7.3 Linux. Unless indicated otherwise, a Hi-Speed USB flash memory keychain was used. Clients use VMware Workstation 3.1 and have an 8 GB Coda file cache. The VM is configured to have 256 MB of RAM and 4 GB of disk, and runs Windows XP as the guest OS. As in the previous benchmark, we use the NISTNet network emulator

	No Lookaside	With Lookaside	Win
100 Mb/s	14 (0.5)	13 (2.2)	7.1%
10 Mb/s	39 (0.4)	12 (0.5)	69.2%
1 Mb/s	317 (0.3)	12 (0.3)	96.2%
100 Kb/s	4301 (0.6)	12 (0.1)	99.7%

This table shows the resume latency (in seconds) for the CDA benchmark at different bandwidths, with and without lookaside to a USB flash memory keychain. Each data point is the mean of three trials; standard deviations are in parentheses.

Figure 7. Resume Latency

to control bandwidth.

6.2.3 Results

From a user's perspective, the key performance metrics of ISR can be characterized by two questions:

- *How slow is the resume step?*
  This speed is determined by the time to fetch and decompress the physical memory image of the VM that was saved at suspend. This is the smallest part of total VM state that must be present to begin execution. The rest of the state can be demand-fetched after execution resumes. We refer to the delay between the resume command and the earliest possible user interaction as *Resume Latency.*
- *After resume, how much is work slowed?*
  The user may suffer performance delays after resume due to file cache misses triggered by his VM interactions. The metric we use to reflect the user's experience is the total time to perform all the operations in the CDA benchmark (this excludes user think time). We refer to this metric as *Total Operation Latency.*

Portable storage can improve both resume latency and total operation latency. Figure 7 presents our results for the case where a USB flash memory keychain is updated at suspend with the minimal state needed for resume. This is a single 41 MB file corresponding to the compressed physical memory image of the suspended virtual machine. Comparing the second and third columns of this figure, we see that the effect of lookaside caching is noticeable below 100 Mb/s, and is dramatic at 100 Kb/s. A resume time of just 12 seconds rather than 317 seconds (at 1 Mb/s) or 4301 seconds (at 100 Kb/s) can make a world of a difference to a user with a few minutes of time in a coffee shop or a waiting room. Even at 10 Mb/s, resume latency is a factor of 3 faster (12 seconds rather than 39 seconds). The user only pays a small price for these substantial gains: he has to carry a portable storage device, and has to wait for the device to be updated at suspend. With a Hi-Speed USB device this wait is just a few seconds.

	No Lookaside	With Lookaside	Win
100 Mb/s	173 (9)	161 (28)	6.9%
10 Mb/s	370 (14)	212 (12)	42.7%
1 Mb/s	2688 (39)	1032 (31)	61.6%
100 Kb/s	30531 (1490)	9530 (141)	68.8%

This table gives the total operation latency (in seconds) for the CDA benchmark at different bandwidths, with and without lookaside to a DVD. Each data point is the mean of three trials, with standard deviation in parentheses. Approximately 50% of the client cache misses were satisfied by lookaside on the DVD. The files on the DVD correspond to the image of a freshly-installed virtual machine, prior to user customization.

Figure 8. Total Operation Latency

To explore the impact of lookaside caching on total operation latency, we constructed a DVD with the VM state captured after installation of Windows XP and the Microsoft Office suite, but before any user-specific or benchmark-specific customizations. We used this DVD as a lookaside device after resume. In a real-life deployment, we expect that an entity such as the computing services organization of a company, university or ISP would create a set of VM installation images and matching DVDs for its clientele. Distributing DVDs to each ISR site does not compromise ease of management because misplaced or missing DVDs do not hurt correctness. A concerned user could, of course, carry his own DVD.

Figure 8 shows that lookaside caching reduces total operation latency at all bandwidths, with the reduction being most noticeable at low bandwidths. Figure 12 shows the distribution of *slowdown* for individual operations in the benchmark. We define slowdown as $(T_{BW} - T_{NoISR})/T_{NoISR}$, with T_{BW} being the benchmark running time at the given bandwidth and T_{NoISR} its running time in VMware without ISR. The figure confirms that lookaside caching reduces the number of operations with very large slowdowns.

6.3 Trace Replay

6.3.1 Benchmark Description

Finally, we used the trace replay benchmark described by Flinn *et al.* [6] in their evaluation of data staging. This benchmark consists of four traces that were obtained from single-user workstations and that range in collection duration from slightly less than 8 hours to slightly more than a day. Figure 9 summarizes the attributes of these traces. To ensure a heavy workload, we replayed these traces as fast as possible, without any filtering or think delays.

6.3.2 Experimental Setup

The experimental setup used was the same as that described in Section 6.1.2.

Trace	Bandwidth	Lookaside Device State			
		No Device	100%	66%	33%
Purcell	100 Mb/s	50.1 (2.6)	53.1 (2.4)	50.5 (3.1)	48.8 (1.9)
	10 Mb/s	61.2 (2.0)	55.0 (6.5)	56.5 (2.9)	56.6 (4.6)
	1 Mb/s	292.8 (4.1)	178.4 (3.1)	223.5 (1.8)	254.2 (2.0)
	100 Kb/s	2828.7 (28.0)	1343.0 (0.7)	2072.1 (30.8)	2404.6 (16.3)
Messiaen	100 Mb/s	26.4 (1.6)	31.8 (0.9)	29.8 (0.9)	27.9 (0.8)
	10 Mb/s	36.3 (0.5)	34.1 (0.7)	36.7 (1.5)	37.8 (0.5)
	1 Mb/s	218.9 (1.2)	117.8 (0.9)	157.0 (0.6)	184.8 (1.3)
	100 Kb/s	2327.3 (14.8)	903.8 (1.4)	1439.8 (6.3)	1856.6 (89.2)
Robin	100 Mb/s	30.0 (1.6)	34.3 (3.1)	33.1 (1.2)	30.6 (2.1)
	10 Mb/s	37.3 (2.6)	33.3 (3.8)	33.8 (2.5)	37.7 (4.5)
	1 Mb/s	229.1 (3.4)	104.1 (1.3)	143.2 (3.3)	186.7 (2.5)
	100 Kb/s	2713.3 (1.5)	750.4 (5.4)	1347.6 (29.6)	2033.4 (124.6)
Berlioz	100 Mb/s	8.2 (0.3)	8.9 (0.2)	9.0 (0.3)	8.8 (0.2)
	10 Mb/s	12.9 (0.8)	9.3 (0.3)	9.9 (0.4)	12.0 (1.6)
	1 Mb/s	94.0 (0.3)	30.2 (0.6)	50.8 (0.3)	71.6 (0.5)
	100 Kb/s	1281.2 (54.6)	216.8 (0.5)	524.4 (0.4)	1090.5 (52.6)

The above results show how long it took for each trace to complete at different portable device states as well as different bandwidth settings. The column labeled "No Device" shows the time taken for trace execution when no portable device was present and all data had to be fetched over the network. The column labeled 100% shows the results when all of the required data was present on the storage device and only meta-data (i.e. `stat` information) was fetched across the network. The rest of the columns show the cases where the lookaside device had varying fractions of the working set. Each data point is the mean of three trials; standard deviations are in parentheses.

Figure 10. Time for Trace Replay

Trace	Number of Operations	Length (Hours)	Update Ops.	Working Set (MB)
purcell	87739	27.66	6%	252
messiaen	44027	21.27	2%	227
robin	37504	15.46	7%	85
berlioz	17917	7.85	8%	57

This table summarizes the file system traces used for the benchmark described in Section 6.3. "Update Ops." only refer to the percentage of operations that change the file system state such as *mkdir*, *close-after-write*, etc. but not individual reads and writes. The working set is the size of the data accessed during trace execution.

Figure 9. Trace Statistics

6.3.3 Results

The performance metric in this benchmark is the time taken for trace replay completion. Although no think time is included, trace replay time is still a good indicator of performance seen by the user.

To evaluate the performance in relation to the portable device state, we varied the amount of data found on the device. This was done by examining the pre-trace snapshots of the traced file systems and then selecting a subset of the trace's working set. For each trace, we began by randomly selecting 33% of the files from the pre-trace snapshot as the initial portable device state. Files were again randomly added to raise the percentage to 66% and then finally 100%. However, these percentages do not necessarily mean that the data from every file present on the portable storage device was used during the benchmark. The snapshot creation tool also creates files that might be overwritten, unlinked, or simply `stat`-ed. Therefore, while these files might be present on the portable device, they would not be read from it during trace replay.

Figure 10 presents our results. The baseline for comparison, shown in column 3 of the figure, was the time taken for trace replay when no lookaside device was present. At the lowest bandwidth (100 Kb/s), the win due to lookaside caching with an up-to-date device was impressive: ranging from 83% for the Berlioz trace (improving from 1281.2 seconds to 216.8 seconds) to 53% for the Purcell trace (improving from 2828.7 seconds to 1343.0 seconds). Even with devices that only had 33% of the data, we were still able to get wins ranging from 25% for the Robin trace to 15% for the Berlioz and Purcell traces.

At a bandwidth of 1 Mb/s, the wins still remain substantial. For an up-to-date device, they range from 68% for the Berlioz trace (improving from 94.0 seconds to 30.2 seconds) to 39% for the Purcell trace (improving from 292.8 seconds to 178.4 seconds). Even when the device contain less useful data, the wins still range from 24% to 46% when the device has 66% of the snapshot and from 13% to 24% when the device has 33% of the snapshot.

On a slow LAN (10 Mb/s) the wins can be strong for an up-to-date device: ranging from 28% for the Berlioz trace (improving from 12.9 seconds to 9.3 seconds) to 6% for Messiaen (improving from 36.3 seconds to 34.1 seconds). Wins tend to tail off beyond this point as the device contains lesser fractions of the working set but it is important to note that performance is never significantly below that of the baseline.

Only on a fast LAN (100 Mb/s) does the overhead of lookaside caching begin to dominate. For an up-to-date device, the traces show a loss ranging from 6% for Purcell (changing from 50.1 seconds to 53.1 seconds) to a loss of 20% for Messiaen (changing from 26.4 seconds to 31.8 seconds). While the percentages might be high, the absolute difference in number of seconds is not and might be imperceptible to the user. It is also interesting to note that the loss decreases when there are fewer files on the portable storage device. For example, the loss for the Robin trace drops from 14% when the device is up-to-date (difference of 4.3 seconds) to 2% when the device has 33% of the files present in the snapshot (difference of 0.6 seconds). As mentioned earlier in Section 6.1.3, the system should suppress lookaside in such scenarios.

Even with 100% success in lookaside caching, the 100 Kb/s numbers for all of the traces are substantially greater than the corresponding 100 Mb/s numbers. This is due to the large number of meta-data accesses, each incurring RPC latency.

7 Broader Uses of Lookaside Caching

Although motivated by portable storage, lookaside caching has the potential to be applied in many other contexts. Any source of data that is hash-addressable can be used for lookaside. Distributed hash tables (DHTs) are one such source. There is growing interest in DHTs such as Pastry [23], Chord [29], Tapestry [33] and CAN [22]. There is also growing interest in planetary-scale services such as PlanetLab [19] and logistical storage such as the Internet Backplane Protocol [2]. Finally, hash-addressable storage hardware is now available [5]. Together, these trends suggest that *Content-Addressable Storage (CAS)* will become a widely-supported service in the future.

Lookaside caching enables a conventional distributed file system based on the client-server model to take advantage of the geographical distribution and replication of data offered by CAS providers. As with portable storage, there is no compromise of the consistency model. Lookaside to a CAS provider improves performance without any negative consequences.

We have recently extended the prototype implementation described in Section 5 to support off-machine CAS providers. Experiments with this extended prototype confirm its performance benefits. For the ISR benchmark described in Section 6.2, Figure 11 shows the performance benefit of using a LAN-attached CAS provider with same contents as the DVD of Figure 8. Since the CAS provider is on a faster machine than the file server, Figure 11 shows a substantial benefit even at 100 Mb/s.

Another potential application of lookaside caching is in implementing a form of *cooperative caching* [1, 4]. A collection of distributed file system clients with mutual trust (typically at one location) can export each other's file caches

	No Lookaside	With Lookaside	Win
100 Mb/s	173 (9)	103 (3.9)	40.1%
10 Mb/s	370 (14)	163 (2.9)	55.9%
1 Mb/s	2688 (39)	899 (26.4)	66.6%
100 Kb/s	30531 (1490)	8567 (463.9)	71.9%

This table gives the total operation latency (in seconds) for the CDA benchmark of Section 6.2 at different bandwidths, with and without lookaside to a LAN-attached CAS provider. The CAS provider contains the same state as the DVD used for the results of Figure 8. Each data point is the mean of three trials, with standard deviation in parentheses.

Figure 11. Off-machine Lookaside

as CAS providers. No protocol is needed to maintain mutual cache consistency; divergent caches may, at worst, reduce lookaside performance improvement. This form of cooperative caching can be especially valuable in situations where the clients have LAN connectivity to each other, but poor connectivity to a distant file server. The heavy price of a cache miss on a large file is then borne only by the first client to access the file. Misses elsewhere are serviced at LAN speeds, provided the file has not been replaced in the first client's cache.

8 Conclusion

"Sneakernet," the informal term for manual transport of data, is alive and well today in spite of advances in networking and distributed file systems. Early in this paper, we examined why this is the case. Carrying your data on a portable storage device gives you full confidence that you will be able to access that data anywhere, regardless of network quality, network or server outages, and machine configuration. Unfortunately, this confidence comes at a high price. Remembering to carry the right device, ensuring that data on it is current, tracking updates by collaborators, and guarding against loss, theft and damage are all burdens borne by the user. Most harried mobile users would gladly delegate these chores if only they could be confident that they would have easy access to their critical data at all times and places.

Lookaside caching suggests a way of achieving this goal. Let the true home of your data be in a distributed file system. Make a copy of your critical data on a portable storage device. If you find yourself needing to access the data in a desperate situation, just use the device directly — you are no worse off than if you were relying on sneakernet. In all other situations, use the device for lookaside caching. On a slow network or with a heavily loaded server, you will benefit from improved performance. With network or server outages, you will benefit from improved availability if your distributed file system supports disconnected operation and if you have hoarded all your meta-data.

Notice that you make the decision to use the device di-

rectly or via lookaside caching at the point of use, not *a priori*. This preserves maximum flexibility up front, when there may be uncertainty about the exact future locations where you will need to access the data. Lookaside caching thus integrates portable storage devices and distributed file systems in a manner that combines their strengths. It preserves the intrinsic advantages of performance, availability and ubiquity possessed by portable devices, while simultaneously preserving the consistency, robustness and ease of sharing/collaboration provided by distributed file systems.

One can envision many extensions to lookaside caching. For example, the client cache manager could track portable device state and update stale files automatically. This would require a binding between the name space on the device and the name space of the distributed file system. With this change, a portable device effectively becomes an extension of the client's cache. Another extension would be to support lookaside on individual blocks of a file rather than a whole-file basis. While this is conceptually more general, it is not clear how useful it would be in practice because parts of files would be missing if the portable device were to be used directly rather than via lookaside.

Overall, we believe that the current design of lookaside caching represents a sweet spot in the space of design tradeoffs. It is conceptually simple, easy to implement, and tolerant of human error. It provides good performance and availability benefits without compromising the strengths of portable storage devices or distributed file systems. A user no longer has to choose between distributed and portable storage. You can cache as well as carry!

9 Acknowledgments

We would like to thank Shafeeq Sinnamohideen for implementing the support for off-machine CAS providers and Casey Helfrich for his help with the Internet Suspend/Resume experiments.

References

[1] ANDERSON, T., DAHLIN, M., NEEFE, J., PATTERSON, D., ROSELLI, D., WANG, R. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating System Principles* (Copper Mountain, CO, December 1995).

[2] BECK, M., MOORE, T., PLANK, J.S. An End-to-End Approach to Globally Scalable Network Storage. In *Proceedings of the ACM SIGCOMM Conference* (Pittsburgh, PA, August 2002).

[3] BOLOSKY, W.J., DOUCEUR, J.R., ELY, D., THEIMER, M. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Proceedings of the ACM SIGMETRICS Conference* (Santa Clara, CA, June 2000).

[4] DAHLIN, M.D., WANG, R.Y., ANDERSON, T.E., PATTERSON, D.A. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the USENIX 1994 Operating Systems Design and Implementation Conference* (Monterey, CA, November 1994).

[5] EMC CORP. *EMC Centera Content Addressed Storage System*, 2003. http://www.emc.com/.

[6] FLINN, J., SINNAMOHIDEEN, S., TOLIA, N., AND SATYANARAYANAN, M. Data Staging on Untrusted Surrogates. In *Proceedings of the FAST 2003 Conference on File and Storage Technologies* (March 31 - April 2, 2003).

[7] HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[8] HUGHES, J.F., THOMAS, B.W. *Novell's Guide to NetWare 6 Networks*. John Wiley & Sons, 2002.

[9] KOZUCH, M., SATYANARAYANAN, M. Internet Suspend/Resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications* (Calicoon, NY, June 2002).

[10] MANBER, U. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference* (San Fransisco, CA, January 1994), pp. 1–10.

[11] MENEZES, A.J., VAN OORSCHOT, P.C., VANSTONE, S.A. *Handbook of Applied Cryptography*. CRC Press, 2001.

[12] MIGO, FORWARD SOLUTIONS. http://4migo.com/.

[13] MOGUL, J. C., CHAN, Y. M., AND KELLY, T. Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP. In *Proceedings of the First Symposium on Networked Systems Design and Implementation* (San Francisco, CA, March 2004).

[14] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Chateau Lake Louise, Banff, Canada, Oct. 2001).

[15] NIST. Secure Hash Standard (SHS). In *FIPS Publication 180-1* (1995).

[16] NIST NET. http://snad.ncsl.nist.gov/itg/nistnet/.

[17] OLSON, M.A., BOSTIC, K., SELTZER, M. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference* (Monterey, CA, June 1999).

[18] PALUSKA, J. M., SAFF, D., YEH, T., AND CHEN, K. Footloose: A Case for Physical Eventual Consistency and Selective Conflict Resolution. In *Proceedings of the Fifth IEEE Workshop on Mobile Computing Systems and Applications* (Monterey, October 2003).

[19] PETERSON, L., ANDERSON, T., CULLER, D., ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the First ACM Workshop on Hot Topics in Networks* (Princeton, NJ, October 2002).

[20] QUINLAN, S., AND DORWARD, S. Venti: A New Approach to Archival Storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies* (January 2002).

[21] RABIN, M. Fingerprinting by Random Polynomials. In *Harvard University Center for Research in Computing Technology Technical Report TR-15-81* (1981).

[22] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., SHENKER, S. A Scalable Content-addressable Network. In *Proceedings of the ACM SIGCOMM Conference* (San Diego, CA, August 2001).

[23] ROWSTRON, A., DRUSCHEL, P. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (Heidelberg, Germany, November 2001).

[24] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., LYON, B. Design and Implementation of the Sun Network File System. In *Summer Usenix Conference Proceedings* (Portland, OR, June 1985).

[25] SOBTI, S., GARG, N., ZHENG, F., LAI, J., SHAO, Y., ZHANG, C., ZISKIND, E., KRISHNAMURTHY, A., AND WANG, R. Segank: A Distributed Mobile Storage System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies* (San Francisco, CA, March 31 - April 2, 2004).

[26] SOBTI, S., GARG, N., ZHANG, C., YU, X., KRISHNAMURTHY, A., WANG, R. PersonalRAID: Mobile Storage for Distributed and Disconnected Computers. In *Proceedings of the First USENIX Conference on File and Storage Technologies* (Monterey, CA, Jan 2002).

[27] SPRING, N. T., AND WETHERALL, D. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of ACM SIGCOMM* (August 2000).

[28] STEINER, J.G., NEUMAN, C., SCHILLER, J.I. Kerberos: An Authentication Service for Open Network Systems. In *USENIX Conference Proceedings* (Dallas, TX, Winter 1988).

[29] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M.F., BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001* (San Diego, CA, August 2001).

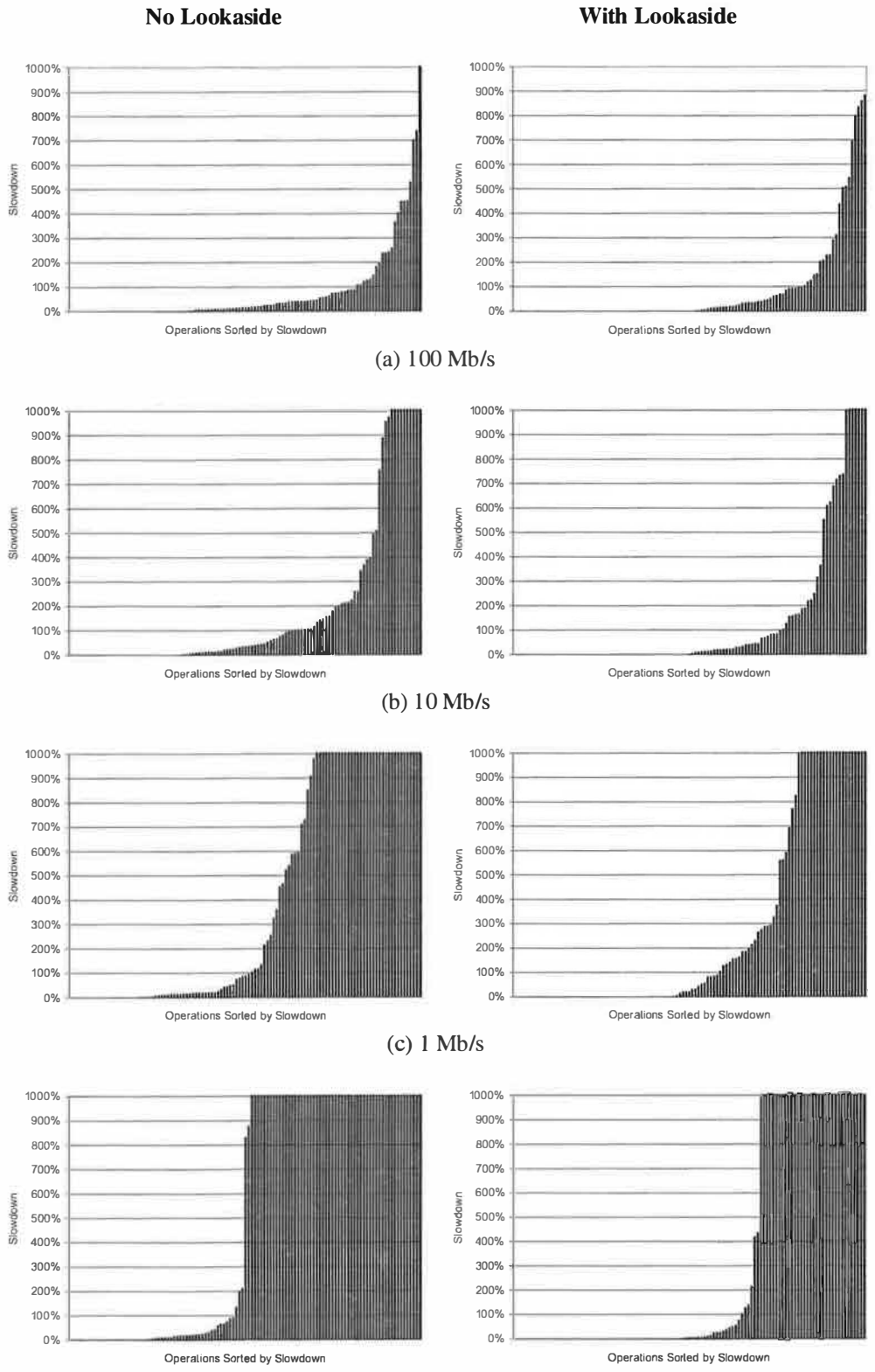
[30] TERRY, D.B. Caching Hints in Distributed Systems. *IEEE Transactions on Software Engineering 13*, 1 (January 1987).

[31] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., BRESSOUD, T., PERRIG, A. Opportunistic Use of Content-Addressable Storage for Distributed File Systems. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, TX, June 2003).

[32] WANT, R., PERING, T., DANNEELS, G., KUMAR, M., SUNDAR, M., AND LIGHT, J. The Personal Server: Changing the Way We Think About Ubiquitous Computing. In *Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing* (Goteborg, Sweden, 2002).

[33] ZHAO, B.Y., KUBATOWICZ, J., JOSEPH, A. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Tech. Rep. UCB/CSD-01-1141, University of California at Berkeley, April 2001.

[34] ZWICKY, E.D., COOPER, S., CHAPMAN, D.B. *Building Internet Firewalls, Second Edition.* O'Reilly & Associates, Inc., 2000, ch. 17.4: File Sharing for Microsoft Networks.



These figures compares the distribution of slowdown for the operations of the CDA benchmark without lookaside caching to their slowdowns with lookaside caching to a DVD.

Figure 12. Impact of Lookaside Caching on Slowdown of CDA Benchmark Operations

Segank: A Distributed Mobile Storage System

Sumeet Sobti* Nitin Garg* Fengzhou Zheng* Junwen Lai* Yilei Shao*
Chi Zhang* Elisha Ziskind* Arvind Krishnamurthy† Randolph Y. Wang*

Abstract

This paper presents a distributed mobile storage system designed for storage elements connected by a network of non-uniform quality. Flexible data placement is crucial, and it leads to challenges for locating data and keeping it consistent. Our system employs a location- and topology-sensitive multicast-like solution for locating data, lazy peer-to-peer propagation of invalidation information for ensuring consistency, and a distributed snapshot mechanism for supporting sharing. The combination of these mechanisms allows a user to make the most of what a non-uniform network has to offer in terms of gaining fast access to fresh data, without incurring the foreground penalty of keeping distributed elements on a weak network consistent.

1 Introduction

In this paper, we study the construction of a mobile storage system designed to work on distributed storage elements connected by a network of non-uniform quality. The target environment of our system is one where all storage elements are connected with each other, but only some storage elements, typically those that are close to each other, enjoy high-quality links.

1.1 The Target Environment

This non-uniformly connected world is the reality today and it is continuing to evolve. There are three aspects of this development. First, low-cost short-range wireless technologies, such as 802.11 and Bluetooth, are proliferating, which allow mobile elements in a small neighborhood to be spontaneously connected with each other at a level of quality that is quite good. Second, when a fast WiFi "gateway" into the Internet is not available, pervasive but low-quality wireless connectivity in the wide area using technologies such as cellular modems has become very affordable. Third, stationary storage elements are becoming increasingly wired and they are "always on" the network. These may include not only computers in offices and server rooms, but also broadband-connected computers at home and hotels, and an increasing array of entertainment appliances such as Tivo-like personal video recorders. The connectivity quality among these devices also exhibits a high degree of variance. A typical DSL-connected home computer, for example, may only have an up-link capacity around 100 Kbps.

Despite the high variance in connectivity quality, total disconnection is (or can be) increasingly rare, as those who own BlackBerry email devices and those who experiment with Internet access on transcontinental flights are beginning to realize. While our system has provisions to cope with it, total disconnection is *not* our top focus. Instead, our focus is to cope with a non-uniform but always-on interconnect linking distributed and mobile storage elements.

1.2 Requirements

We begin by considering some example usage scenarios. A user owns several computers. Perhaps some of them are in his office, some at his DSL-linked home, and some in an "off-site" office in a different city, which he occasionally visits. Some of them are desktop machines, and others are laptops and PDAs that may accompany the user when he travels.

When the user arrives at his office, some of his latest work may have been done on a laptop that he carried home the night before and is still with him. At this time, the user should not be forced to wait for all the new data to propagate from the laptop to the office desktop before he is allowed to resume work on the desktop. He should be able to see and operate on the complete and latest view of his data from his desktop immediately. Also, the user should not have to remember where the latest copy of a particular piece of data is.

The next day, the user may run into a colleague on a train and the two spontaneously decide to share some files. In this case, the system should try its best to satisfy the requests using the ad hoc 802.11 link between the two laptops, and resort to a cellular modem to reach data

*Department of Computer Science, Princeton University, {sobti, nitin, zheng, lai, yshao, chizhang, eziskind, rywang}@cs.princeton.edu.

†Department of Computer Science, Yale University, arvind@cs.yale.edu.

Krishnamurthy is supported by NSF grants CCR-9985304, ANI-0207399, and CCR-0209122. Wang is supported by NSF grants CCR-9984790 and CCR-0313089.

that is only available at the office or home. On a third day, when the user is again on the train, and a colleague in the office tries to read some of his files, the system would instead attempt to satisfy the colleague's requests using a copy stored on the office LAN, on a DSL-linked home machine, or on the cellular modem-connected laptop on the train, in that order of preference.

We summarize the requirements of the system. In our target system, data may be stored on, moved to, and replicated at any device for performance optimization and reliability purposes. No device necessarily houses all the data. A user sees a single image of name space spanning all the devices. A user experiences coherent semantics even when he sends read and write requests into the system from different entrance points of the network. Data and metadata propagations can happen in the background; but no foreground propagation is mandatory for the user to be able to start using a consistent system immediately. An additional requirement that we desire to fulfill is to provide a storage or file system-level solution that can transparently cater to most existing applications.

1.3 The Segank System

We call our system Segank (pronounced *see-gank*). It must solve three key problems: (1) how does the system locate data that can be stored on any devices and how does it choose a best replica? (2) without costly mandatory propagation, how does the system ensure consistency across multiple devices as old data on these devices becomes obsolete? (3) how does the system ensure a consistent image across all devices for the purpose of sharing and backup?

Segank solves the first problem using a location- and topology-sensitive multicast-like solution (Section 3). The advantage of this solution is that it minimizes global state, allows autonomous data movement decisions, and can effectively exploit locality. The system solves the second problem using lazy peer-to-peer propagation of invalidation information (Section 4). The combination of the laziness element and the decoupling of the propagation of invalidation information from that of data minimizes the cost of bringing weakly connected devices up to date. The system solves the third problem using a distributed snapshot mechanism (Section 5). This solution allows one to flexibly trade off freshness of data against performance when facing weak connectivity.

2 Background

2.1 Naive Approaches

Solutions that indiscriminately tax a weak wide area connection are unlikely to be adequate, at least in the foreseeable future. The much anticipated 3G wireless networks, for example, are designed to ultimately achieve 384 Kbps, but industry observers agree that wide availability of such speeds is many years away. Today, most US 3G users can realistically expect data speeds of somewhere between 40 to 80 Kbps, a far cry from the hypothetical speeds of 144 Kbps and 192 Kbps [24]. Two users who meet on a train, for example, are unlikely to be able to communicate and collaborate productively by separately connecting to a remote stationary file-server via weak WAN connections.

The other extreme approach is to avoid using networks altogether. Instead, a user would rely exclusively on a mobile storage device to carry all of his data. This approach, however, is also unlikely to be adequate for several reasons. First, despite the capacity improvement of storage devices, the nature of new applications' appetite for storage is such that the capacity of a single portable device is unlikely to be sufficient for all of a user's storage needs. The capacity of the mobile devices is likely to continue to lag behind that of their stationary counterparts, and we expect much data, such as TV programs recorded on a "Tivo," to continue to reside on these stationary devices. Second, mobile storage devices tend to have poorer performance compared to desktop versions due to considerations such as energy consumption, noise, and form factor. Last, but not least, storage devices, by themselves, provide little support for transparent data sharing among collaborating users.

2.2 Existing Systems

To understand the different challenges posed by non-uniform connectivity and disconnection, let us start by considering the Bayou system [18, 22]. Each Bayou device houses a complete replica of a database, and alternates between two distinct states of operation: "disconnected" and "merging." In the disconnected state, the user of the device only "sees" local state stored on this device. In the merging state, the device communicates with a peer device, and new updates made on each are played onto the other.

While the Bayou model may make sense in a disconnected environment, it is less appropriate for a non-uniform network of storage elements. First, the requirement of housing complete replicas on each device may be unnecessary, expensive, and in some cases, even infeasible. Second, being required to work on a device in a "disconnected" mode is overly restrictive when (potentially fresher) data stored on other devices could have been made available over a network. In Bayou, the only way to access data on other devices is to perform a merge operation with them, play their updates onto the local device, and then read data from the local device. Merging can be time-consuming as it propagates both meta-data and data, and forcing a user to wait until merging finishes can be inconvenient.

While the initial Coda system [11] shares Bayou's disconnected model of operation, later enhancements extend the system to work with a weak network [13]. The more serious problem with Coda is its lack of support of peer-to-peer interaction. Coda differentiates "clients" from "servers" and peer clients do not communicate with each other directly. Each data item has a fixed "home" on the server and clients are always required to "reintegrate" their updates back to the server. Requiring nearby devices to communicate only with a far-away server becomes too strict a constraint when peer-to-peer interactions could have worked well. One additional disadvantage that Coda shares with Bayou is its potential high cost of "merging:" any updates must be played to a server before they become visible to other Coda clients.

A class of existing file and storage systems that do address the missing elements of Coda and Bayou are the peer-to-peer systems: they do not require any machine to house a complete replica; they take advantage of an always-on network; they allow peer-to-peer interactions; and they do not mandate expensive propagations. They, however, exhibit their own problems when exposed to a mobile environment, a context that they have not been designed for. A key problem that a system like Gnutella [6] fails to address is consistency. For example, a mobile user may issue read and write requests into the network of devices from different entrance points, and the user is not guaranteed a consistent view, as data copies of different levels of freshness may coexist in different parts of the network. More recent wide-area peer-to-peer file systems employ distributed hash table-based (DHT-based) placement algorithms [2, 14, 20]. One problem with this approach is that the hash algorithms dictate the placement of data, whereas in our target environment, we need to be able to control data placement and replication in a more flexible manner.

Cluster file systems allow data to be stored flexibly in a fast LAN [1, 12, 23, 16]. These networks, however, have a simple, homogeneous topology that behaves more like a storage backplane, allowing these systems to freely manipulate cohesive distributed data structures. In our target environments, we must exercise care not to overuse weak networks. Data structures that are carelessly spread across many nodes separated by slow links, for example, are unacceptable. Also, the work of keeping distributed storage elements consistent needs to be pushed to the background as much as possible.

2.3 Minimizing Foreground Propagations

Segank allows data stored on distributed devices or owned by different users to be used without mandating expensive foreground propagation among different devices. This is one of the key features that differentiates Segank from systems such as Coda and Bayou. A Bayou client relies exclusively on a single device to satisfy its read requests. Similarly, a Coda client relies on its hoard (and the server in later enhancements). In order to "see" new data written by other clients, mandatory propagation of all updates must occur to bring these devices up-to-date. Such propagation can be expensive on a weakly connected network. A Segank data consumer, on the other hand, is not dependent on any single device. Segank provides a fresh and consistent view of the entire system even when none of the individual devices is entirely "fresh" by itself.

3 Reading Data Using Segankast

Upon a read request, Segank needs to find out which devices have the desired data, and it needs to choose a device to retrieve the data from. In this section, we discuss the data location mechanism. We consider a single reader in this section and defer the discussion of multi-user sharing to Section 5.

A Segank user carries a small device that we call a MOAD (MObile Air-linked Disk.) Transparent to the user, the device plays four roles: (1) storing small amounts of invalidation information that can be quickly accessed to guarantee a consistent view of the system; (2) optionally caching and propagating data to improve performance; (3) providing short-range WiFi connectivity to peer devices (via 802.11 or Bluetooth) whenever possible; and (4) providing wide-area connectivity to far-away always-on devices (via a cellular modem) as a last resort when faster connectivity is not available. We conjecture that an industrial strength version of the MOAD can be packaged in a form factor that is not much larger than a wrist watch. There is, however, nothing special about the hardware requirements of a MOAD, and a PDA or a laptop can serve as a MOAD if it has the required communication capabilities. In our prototype, a Compaq iPAQ equipped with an IBM 1 GB Microdrive is used as the MOAD.

3.1 Drawbacks of Location Maps

One plausible solution to the data location problem is to maintain a mapping from an object ID to a list of devices where a replica of the object can be found. The map itself is too large to be stored on any one device or to be replicated on all devices, so the map needs to be distributed. To cope with a non-uniform network, the system needs to be able to store and replicate pieces of the map as flexibly as it does data; so a higher-level map of map is needed. This leads to a hierarchical map solution where the highest level map should be compact and perhaps easier to manage.

This approach, however, has its drawbacks. Any

data movements, such as caching data at, pushing data to, and evicting cached copies from devices, require reading and/or updating the multi-leveled location map. These operations may involve significant complexity as the system must exercise care to keep various pieces of distributed state consistent with each other. These operations may also introduce extra costs associated with extra network messages and I/Os. Furthermore, the location map approach, in itself, does not answer the question of which copy to actually read when there is more than one to choose from.

3.2 Segankast

Segank does not use location maps. Instead, it employs a mechanism that is similar to multicast: the system queries a number of devices until it locates one that has the desired data. We call this mechanism Segankast. Segankast is different from the data location mechanism used in Gnutella [6] in two important ways: it guarantees a consistent view of the system as a mobile user reads and writes at different locations of the network (described in Section 4.2); and it carefully controls the order, type, and parallelism of the requests to optimize performance (described in Section 3.3).

Segankast has several advantages over the use of location maps for our purposes. The system may freely place, move, replicate, or purge data on any device without having to update location information stored elsewhere. Each device is therefore autonomous. There is no risk of data and its map becoming inconsistent with respect to each other, and there are no complications resulting from, for example, attempting to access map information that is stored farther away than data, or map information that is unreachable although the corresponding data is.

Like user-level multicast systems, Segankast requests are issued over an overlay tree rooted at the current reader device. The tree includes only the devices owned by a single user. We do not envision this number of devices in a single tree to be massive. The tree is location-sensitive, so when a device is at a new location, a new tree rooted at the device is constructed. Figure 1 shows a sample Segankast tree.

3.3 Optimizing Segankast Performance

There are two types of potential performance cost. The first is the latency incurred querying devices that do not contain the desired data. Segankast must carefully control the ordering of its queries. For example, if the desired data is found across a wide-area or a modem link, the extra time spent querying devices on a nearby fast LAN is relatively insignificant. The second type of potential cost is the network contention resulting from multiple data replies. It should be noted, however, that

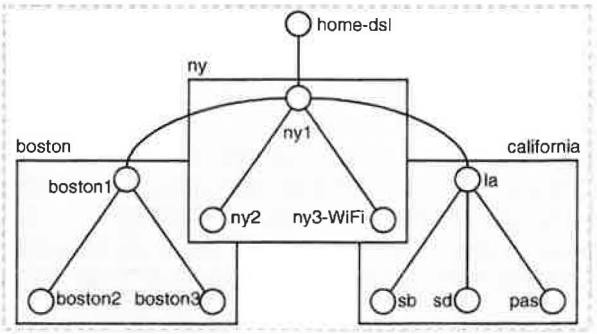


Figure 1: An example Segankast tree. A rectangle represents a "cluster" and the circles are machines.

not all types of contention necessarily can lead to visible Segankast cost. For example, suppose three hosts, A, B, and C share a single fast LAN; if A and B send reply data to C in parallel, while C forwards only one reply over a modem link to a requester R, even though A and B generate contention on the LAN, the contention is not visible to R. These two types of potential costs can be traded off against each other: for a small amount of data reply, for example, minimizing the latency of the request is more important than minimizing the contention of the replies, so one may choose to increase the degree of parallelism in Segankast. These goals make the optimization problem faced by Segankast quite different from that faced by traditional user-level multicast systems [9, 4].

The problem of optimizing Segankast performance has two sub-problems. One is determining the structure of the Segankast tree; and the other is deciding how queries are forwarded on the chosen tree. We note that the following Segankast strategies are only heuristics; better solutions may be possible and they remain a research focus.

3.3.1 Construction of Segankast Trees

Trees are constructed based on probing measurements. Tree construction proceeds in two steps. Step one constructs *clusters*, whose members are close to each other and at approximately equal-distance from the single reader device at the root. Each cluster is a subtree. Step two connects the clusters to form a larger tree. The cluster-based two-step approach allows us to simplify the tree construction via divide-and-conquer.

We use the example in Figure 1 to illustrate the heuristics used in tree construction. The example includes a DSL-connected home machine in New York, two LAN-connected clusters in New York and Boston, and a WAN-connected cluster of machines in four cities in California. First, we define a cost function. Let r be the root node. Let $t(r \leftarrow x)$ be the time spent sending a block from x to r directly, and $t(r \leftarrow y \leftarrow x)$ be the

time spent sending a block from x to r via y. We define the *cost* of attaching x to y to be:

$$\frac{t(r \leftarrow y \leftarrow x) - t(r \leftarrow x)}{t(r \leftarrow y)}$$

The numerator represents the penalty incurred by an extra hop (which can be negative if the overlay route is better than the Internet route), and it is normalized against the distance to the intermediate node in the denominator.

In step one, we incrementally form clusters by considering the non-root nodes in increasing order of their latencies from the root node. We begin with a single cluster containing only the node (`ny1`) with the lowest latency from the root (`home-dsl`). Incrementally, we attempt to attach the next node to one of the previously-added nodes. The position of attachment is determined by the cost function. In the example, the costs of attaching `ny2` and `ny3-WiFi` to `ny1` are low, so all these nodes are declared to be in the same cluster (`ny`). If cost of attaching a node to existing clusters is high (exceeds a heuristic threshold), it starts a new cluster. The cost of attaching `la` to any node in the current set of clusters, for example, is high, so it starts a new cluster.

Step two connects clusters to form trees. We build separate trees to optimize for latency and bandwidth. During this step, we only consider the root nodes of the cluster subtrees (`ny1`, `boston1`, and `la` in the example), and the root of the tree (`home-dsl`). To form the Segankast tree designed to optimize latency, we consider the complete graph spanning these nodes. We annotate each edge by the round-trip time of fetching a block between a pair of nodes, and compute the shortest-path-tree rooted at the root node. To form the Segankast tree designed to optimize bandwidth, we annotate each edge of the complete graph by the inverse of the edge bandwidth, and compute the minimum-spanning-tree.

3.3.2 Forwarding Segankast Requests

Upon receiving a request, each node in a Segankast tree has two decisions to make: whether to forward the request in parallel to its children or sequentially, and the type of messages to send. In terms of the second decision, there are two choices: a direct *fetch*, or a *test-and-fetch* that first queries which children (if any) have the desired data and then issues a separate message to retrieve data from a chosen child. These two decisions can be combined to form three viable strategies: (1) parallel fetch, (2) parallel test-and-fetch, and (3) sequential fetch.

We use the following strategy to choose how to forward requests at each node. If the message received from the parent is a parallel test-and-fetch, we simply propagate it in parallel to all children. If simultaneous replies from all children can exceed the bottleneck bandwidth to the reader device, we use parallel test-and-fetch. Otherwise, we use parallel fetch. In the near future, we plan to incorporate location hints of target data, and sequential fetch, which is not currently used in the prototype, may become appropriate.

4 Maintaining Consistency

When data is deleted or overwritten, devices that house obsolete copies need to be "informed" so the storage space can be reclaimed. Furthermore, we must ensure that Segankast does not mistakenly return obsolete data even when a mobile user initiates requests from different entrance points of the network. We desire to achieve these goals without mandating foreground propagations of either data or metadata. In this section, we consider operations involving a single owner/writer, who always has his MOAD device with him. We consider sharing (reading/writing by multiple users) in Section 5.

4.1 Propagating the Invalidation Log

A naive solution is to send invalidation messages to all devices belonging to the owner. Due to the non-uniform network, however, some of the devices may be poorly connected, so foreground invalidation is not always feasible. Lazy invalidation is especially appealing when the quality of connectivity may change significantly due to mobility.

Each write in Segank is tagged with a monotonically incrementing counter, or a *timestamp*. (This is a local counter maintained on the MOAD.) The sequence of write operations in the increasing timestamp order is called the *invalidation log* of the system. Specifically, the invalidation log entry of a write operation contains the ID of the object written and the timestamp. Each device stores its data in a persistent data-structure that we call a *block store*. The meta-data stored with each data object includes the timestamp of the write operation that created the data.

Each device also has a persistent data-structure to store the invalidation log. Segank, however, does not force any device (except a user's MOAD, as we discuss below) to store the complete invalidation log. In fact, the portion of the invalidation log stored on a device may not even be contiguous.

We assume that the MOAD houses the most complete invalidation log as it follows the user (who is the sole writer for the purpose of our present discussion). The head of the log can be truncated once it has been sent to all other devices of this user. The size of the log is bounded by the amount of new data writes performed

in a certain period of time, which should be smaller than that of a location map, since a location map must map all the data in the system. Parts of the log can also be stored on other well-connected devices if the capacity on the MOAD becomes a premium. Since the MOAD is always with the user and it can communicate using at least the wireless modem link, the entire invalidation log should always be reachable. Also, it is easy to turn any other device (a laptop, for example) into a MOAD simply by transferring the invalidation log onto it, provided the device has the same communication capabilities as the MOAD. Therefore, to simplify the rest of the discussion, we assume that the device that a user works on is a MOAD, and that it contains the entire invalidation log.

As the user works on a MOAD device and creates data, new entries are appended to the invalidation log on the device. This new tail of the log is propagated to other devices in the background. Log propagation is a peer-to-peer operation that can happen between any two devices. If a device houses a piece of the log that another lacks, then log propagation can be performed. For efficiency, in the normal case, log propagation is performed along the edges of the Segankast tree, especially those that correspond to high-quality network links. It must be noted, however, that all propagation is performed in the background.

Having received a portion of the log, a device may decide to "play" the log entries onto its block store at any convenient time. Playing a log entry onto the block store means discarding any data that is overwritten by the operation in the entry. For correctness (especially of the snapshot design described in Section 5.2), log entries are played only in strict timestamp order. We define *freshness* of a device to be the timestamp of the last log entry played onto it. Devices other than the MOAD are not expected to store the log forever. Log fragments may be discarded at any time after having been played.

Note that only the invalidation records need to be propagated in the background and no data exchange is necessary to ensure a consistent view of the Segank system. The amount of the invalidation information should be at least three orders of magnitude smaller than that of data. This is in contrast to existing systems where data and metadata propagations are intertwined in the same logs [11, 13, 14, 18, 22].

All the devices in a Segank system are very much similar to each other. One difference between the MOAD and the other devices is that the MOAD is guaranteed to have the most complete invalidation log. (Although as we have said, even this difference is not strictly necessary.) As long as the MOAD is with the user, it ensures fast access to the invalidation information, which as we describe in the next section, is sufficient to ensure a consistent view of the system without mandating any type of foreground propagation.

4.2 Querying Invalidation Logs for Reads

It is not necessary to bring a device up to date by playing the invalidation records on it before it can participate in the Segankast protocol for reads.

Suppose a user is working on his laptop MOAD device with the most complete invalidation log. We maintain a sufficiently long tail of the invalidation log in a hash-table like data-structure that supports the following operation: given an object ID, it locates the last write (and the corresponding timestamp) to that object in the tail. We refer to this portion of the invalidation log as the "queryable log." Upon a read request, the system queries the queryable log to look for the latest write to the requested object. If an entry for the object is found, the system launches a Segankast request, specifically asking for an object with the timestamp found in the queryable log. When any device (including the local device) receives this Segankast query, without regard to its own freshness value, it queries its block store for the object with the specified timestamp. When the data is found on some device, the read request is satisfied.

If no entry for the object is found in the queryable log, it implies that the data has not been overwritten during the entire time period reflected in the queryable log. Suppose the head of the queryable log on the MOAD has a timestamp of t_0. The system then launches a Segankast request asking only devices with freshness at least $t_0 - 1$ to respond.

An invariant of the system is that all the devices reachable by Segankast at this moment must be at least as fresh as $t_0 - 1$. This invariant, however, does not imply that the complete invalidation log must be queryable: older portions of the invalidation log that are kept for currently-unreachable devices need not be queryable. A device can be beyond the reach of Segankast because, for example, it is currently disconnected, in which case there is no danger of reading obsolete data from it. When such a device later becomes reachable again, the system must restore the invariant by either making more of the older portion of the log queryable, or by playing this older portion of the log to the newly connected device to upgrade its freshness up to $t_0 - 1$. This invariant makes it possible to cache the queryable tail of the invalidation log entirely in memory, minimizing overhead paid on reads. Note that this protocol handles disconnection without extra provisions. It also allows the system to flexibly choose how aggressively the invalidation log should be propagated: a weakly-connected device need not receive such propagations while still being able to supply consistent data. The overall effect of this protocol is that a consistent view of the system is always maintained without any mandatory foreground propaga-

tion, even though individual devices are allowed to contain obsolete information.

4.3 Data Movement and Discard

As explained earlier, an important feature of Segank compared to some existing epidemic exchange-based systems is that the propagation of the invalidation records and that of data can be decoupled. Data movement is mostly a performance optimization, and it is largely decided by policy decisions. At one extreme, an aggressive replication policy effectively can also support disconnected operation. One goal of the Segank design is to allow individual devices or subsets of devices to autonomously make data movement/discard decisions without relying on global state or global coordination. There are, however, still some constraints.

One of them concerns data movement: data is only sent from fresher devices to less fresh devices. This constraint ensures that if the propagated data is overwritten, the corresponding invalidation record is guaranteed to not have been played to the data receiver prematurely. Interestingly, there is no constraint on the relationship between the timestamp of the propagated data and the freshness of the receiver device. Another constraint is that we need to exercise care not to discard a last lone copy of the data. We adopt a simple solution: when data is initially created, a *golden copy* is established; and a device is not allowed to discard a golden copy without propagating a replacement golden copy to another device.

5 Sharing with Snapshots

Segank supports sharing using a "snapshot" mechanism—a *snapshot* represents a consistent state of an owner's data "frozen" at one point in time.

5.1 Requirements of Sharing

Let us examine an example scenario. On day 1, users A and B are collaborating in a well-connected office. A may wish to promptly see fresh data being continuously produced on a desktop by B. Over night, some of the data produced by B may be propagated to a laptop of B's. On day 2, B takes his laptop onto a train, where only a cellular modem is available, and he continues to modify some (but not all) of his data.

On day 2, A has many options available to him when accessing B's data in terms of how fresh he desires the data to be. (1) A may decide that the data produced by B on day 1 is fresh enough. In this case, A's read requests are satisfied entirely by B's office desktop without ever using the cellular modem. Again, this is effectively supporting disconnected operation. (2) A may desire to see a snapshot of B's data at, say, noon of day 2. For the data that B has not modified by noon, A's read requests may be satisfied by B's office desktop; but occasionally, A may need to use the cellular modem to access a piece of data produced by B before noon on day 2. (3) A may desire to see a new snapshot of B's data, say, every minute. A now uses the cellular modem more often.

We summarize some requirements. First, consider case (2) above. In order for A to read a piece of data written by B shortly before noon, A should not have to wait for B to flush all the data that B has produced by then. In fact, we should not even necessarily force B to flush its invalidation information. A should be able to read whatever data whenever he desires on demand. In other words, no mandatory propagation of any kind should be necessary to guarantee a certain degree of freshness. This requirement is not limited to case (2)—it is a general property of Segank. Second, facing a non-uniform network, a user should be able to precisely control when and how often a new snapshot is created for the shared data to trade off freshness against performance.

5.2 Snapshots

To support snapshots, we rely on a copy-on-write feature in the block store. A snapshot is created or deleted simply by appending a snapshot creation or deletion record to the invalidation log. These operations are instantaneous. The in-order propagation of the invalidation log among the devices ensures that data overwritten in different snapshots is not inadvertently deleted. Each snapshot is identified by a *snapshot ID* based on a monotonically-increasing counter kept persistent on the MOAD. (This counter is different from the counter used to assign timestamps to write operations.) The snapshot named by the ID contains the writes that have occurred between the creation of the previous snapshot and this snapshot. In the rest of this discussion, the timestamp of a snapshot is understood to be the timestamp of the creation record of the snapshot. Since a snapshot must be internally consistent, the decision as to when to create or delete a snapshot must be made by higher-level software (e.g., the file system) or the user.

5.3 Read Sharing

When user A reads data created by user B, we call A a *foreign reader*. The foreign reader first chooses a snapshot of a desired recentness. To do this, the reader device contacts a device belonging to the writer. To obtain the most recent snapshot ID, the reader must contact the MOAD owned by the writer.

Suppose the reader desires to read an object from a snapshot with timestamp T_S. In the cases where either the devices with the desired object have freshness at least T_S, or where the reader happens to have a long enough tail of the invalidation log ending at T_S, the description

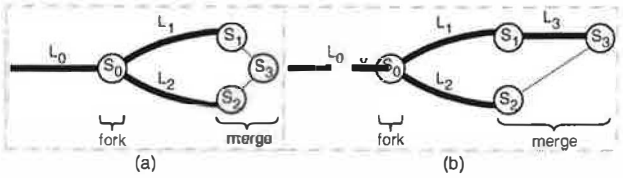


Figure 2: Snapshots for write-sharing. (a) Without conflicts, (b) with conflicts.

of Section 4.2 still applies. In other cases, we need to extend the earlier description.

Recall that the read algorithm given in Section 4.2 requires the reader to query the invalidation log to ensure consistency. This querying yields a timestamp of the desired data if it is written in the period covered by the invalidation log, or the timestamp of the head of the log. In this earlier discussion, when the reader and the writer users are the same, the complete invalidation log is available on the single user's MOAD for fast querying. In the case of a foreign reader, however, this assumption no longer holds. A simple solution is to first query the invalidation log stored on the remote MOAD owned by the writer for the timestamp. Once the reader obtains the timestamp, the rest of the read process remains the same as described earlier. The disadvantage of this approach is that the writer's MOAD device may be weakly connected, and querying it for all reads can be expensive.

To overcome the inefficiency, we note that fragments of the invalidation log may have been propagated to other devices (in the background), some of which may be better connected to the reader device. Indeed, some or all of the invalidation log fragments may have been propagated to the reader device itself. One possible improvement is to split each Segankast into two phases: a first phase queries the devices to obtain the target timestamp in the invalidation log fragments, and a second phase retrieves the data as described earlier. (A modified parallel test-and-fetch strategy given in Section 3.3.2 is best suited for these foreign reads.) In a more sophisticated improvement, it is possible to combine these two phases into a single one in certain cases. Due to lack of space, we omit the details of these improvements.

The mechanism described above ensures that a foreign reader can read a snapshot of certain recentness without waiting for foreground propagation of either data or the invalidation log. However, the data-less invalidation log can be propagated efficiently in the background on most networks (see Section 8). So, by default, Segank aggressively propagates the invalidation log.

5.4 Write Sharing

We start by considering two writers (illustrated in Figure 2.) The two users begin with a single consistent file system. To start concurrent write-sharing, they perform a *fork* operation, which names the initial snapshot as S_0, and allows the two users to concurrently write into two new snapshots, S_1 and S_2, in isolation. New updates by one user are not visible to the other, until when the two users desire to make their new data available to each other by performing a *merge* operation. Prior to the fork, the invalidation log is L_0. Prior to the merge, the two users' new writes result in two separate invalidation log fragments L_1 and L_2.

The first step of the merge operation is conflict detection. This is an application-specific process that should be dependent on the nature of the software running on top of Segank. The three snapshots S_0, S_1, and S_2 are available to the conflict detection process as inputs. In our prototype, we lay a file system on top, and the three snapshots manifest themselves as three distinct file systems. In theory, a possible way of implementing conflict detection in this case is to recursively traverse the three file systems to identify files and directories that have been modified and to determine whether the modifications constitute conflicts. This slow traversal, however, is not necessary. As modifications are made to S_1 and S_2, the higher-level software should have recorded book-keeping information to aid later merging.

In our prototype, we modify the file system running on Segank to capture path names of modified files and directories. This information is summarized as a tree of modification bits that partially mirrors the hierarchical name space. A node in this tree signifies that *some* objects below the corresponding directory are modified. This tree of modification information is logged separately to the MOAD device and is cached. The conflict detection is implemented by comparing the two trees. In what we believe to be the common case of modifications being restricted to a modest number of subdirectories, this comparison can be quickly made even on a weak connection. The amount of information exchanged between the two nodes should be far smaller than the invalidation logs. The details of such an application-specific conflict detection mechanism, however, are not central to the more general Segank system.

If no conflict is found (Figure 2a), the system automatically creates a merged snapshot S_3. Due to the lack of conflicts, the ordering of L_1 and L_2 is irrelevant. The invalidation log resulting in S_3 is logically simply a concatenation of L_0, L_1, and L_2. However, it is not necessary to physically transfer the log fragments among the devices. Since the ordering of log fragments in the system invalidation log is determined by the timestamps of the head entries of these fragments, a simple exchange and reassignment of timestamps is sufficient to effect the "logical concatenation." For instance, the invalidation log for the merged snapshot could be created by retaining the entries from L_1 and assigning timestamps to the

new entries in L_2 such that they logically succeed the tail entry of L_1. The outcome of the merging process is a consistent snapshot and a merged invalidation log that both users can now use to satisfy their read requests in a way similar to what is described in Section 5.3.

If conflicts are found (Figure 2b), an application-specific resolver or user intervention is required. The user is, again, given the three distinct file systems. Suppose the user starts with S_1 and performs a sequence of manual file system modifications to it, incorporating some desired modifications from S_2, and ending up with a new snapshot S_3, the result of resolving conflicts. These new modifications result in a new invalidation log fragment L_3. Although L_1 and L_2 contain invalidation log entries resulting from conflicting updates, their ordering is still unimportant, because any such entries should be superseded by entries in L_3. The outcome of this process is a single consistent invalidation log that is logically the concatenation of L_0, L_1, L_2, and L_3. Again, no transfer of log fragments is necessary to enable either user to read from a consistent snapshot S_3.

The above process can be generalized to more than two concurrent writers by performing repeated pair-wise merging. We note that the process shares some common goals with the read-sharing mechanism. No foreground propagation of data or invalidation is mandatory for the concurrent writers to share consistent snapshots. As a result, in the absence of conflicts, the latencies of all snapshot operations can be kept low, enabling collaborating users to perform these operations frequently when they are well-connected, and to trade off recentness of shared data against performance when they are weakly-connected.

6 Exceptional Events and Limitations

6.1 Disconnected Devices

The Segank system is primarily designed for always-on but non-uniform connectivity: disconnection is not a main focus. Nevertheless, the handling of disconnected devices has been mentioned throughout the previous sections. We now consolidate these descriptions and clarify the limitations. First, possible disconnection and reconnection events cannot cause consistency violations. Neither invalidation log entries nor Segankast requests can reach a disconnected device. Upon reconnection, in order for the previously disconnected device to be eligible to return data, we must first restore an invariant (Section 4.2).

Second, as explained in Section 4.3, how data is moved is a policy decision, decoupled from the basic data location and consistency mechanisms of Segank. An important design goal of the Segank system is to accommodate almost arbitrary policies. Under an aggressive data replication policy tailored for disconnected environments, a Segankast should reach alternative replicas instead of being limited by disconnected copies. Third, if there is not enough time to replicate the freshest data, the snapshot mechanisms described in Section 5 may allow an older consistent snapshot of the system to be read instead. If one exhausts these options, data on a disconnected device would be unavailable until reconnection. Although the Segank system is designed to allow flexible data movement and replication policies, this paper does not provide a systematic study of specific policies designed to minimize potential disruption caused by disconnection and to optimize performance for various usage scenarios and environments. We plan to address this limitation in future research.

6.2 Inaccessible MOAD

As far as the consistency and accessibility of the data located on the MOAD is concerned, the description of Section 6.1 still applies. The MOAD, however, houses the most complete invalidation log, and access to this log is necessary for ensuring consistent access to data on other accessible devices. As discussed in Section 4.1, one way of increasing the availability of the log is to replicate it on other well-connected devices, so that when the MOAD is inaccessible, the complete log can still be accessed. If the complete invalidation log is not sufficiently replicated to be accessible and only an older portion of the log is reachable, one may use this partial log and the snapshot mechanism (Section 5) to access data in an older snapshot. Failing these options, one would be unable to access the Segank system.

6.3 Backup and Restore

At least two factors complicate the handling of device losses. First, because the Segank system can function as a low-level storage system, the loss of a device can result in the loss of a fraction of data blocks managed by the system, making maintaining high-level system integrity challenging. Second, the mobile and weakly connected environments within which Segank operates mean that it may not be always possible to dictate replication of data on multiple devices. Therefore, our goal is not necessarily guaranteeing the survival of each data item at all times, which is not always possible; instead, our goal is to maintain system integrity and preserving as much data as possible when we face device losses. We rely on a backup and restore mechanism for achieving these goals.

The snapshot mechanism (Section 5) can be used to create consistent *backup snapshots* that are incrementally copied to backup devices. At one extreme, each device or each site can have its own backup device; at another extreme, we can have a single backup device

(such as a tape) that is periodically transported from site to site to backup all devices onto a single tape; an intermediate number of backup devices are of course possible as well. During restore, one needs to restore the latest snapshot that has all its participating devices completely backed up. More details of the backup/restore mechanism can be found in a technical report [5]. Note that the set of "backup devices" are not fundamentally different from the other storage devices managed by the Segank system, so for example, even in absence of tape drives, one can designate an arbitrary subset of the regular Segank devices as "backup devices," from which we can recover from the loss of remaining devices (such as the potentially more vulnerable MOAD).

6.4 Other Issues

Due to lack of space, we describe the following issues only briefly; more details can be found in a technical report [5]. Crash recovery in Segank is simple. Communications that can result in modification of persistent storage are made atomic. It is sufficient to recover the block store on a crashed device locally, without concerns of causing inconsistent distributed data structures. Segank appears to the file system built on top as a disk and provides only crash-consistent semantics. The file system must run its own recovery code (such as `fsck`). Adding or removing devices, locating names of devices belonging to a particular user, and access control utilize simple or existing mechanisms whose details we omit in this paper.

7 Implementation

Volumes We have developed the system on Linux. The owner initially makes a slightly modified "ext2" file system on a virtual disk backed by Segank and mounts it. We call each of these file systems a *volume*.

We use a pseudo block device driver that redirects the requests to a user-level process, via up-calls, which implements the Segank functionalities. The unit of data managed by Segank is a block, so it is principally a storage-level solution. Modifications to the ext2 file system are needed for data sharing scenarios in terms of acquiring and releasing snapshots, flushing in-kernel caches, capturing the path names of modified files and directories for detecting conflicting write-sharing, and allocating Inodes and blocks in a way so that allocations by different users do not collide. The block store on each device is a log-structured logical disk [3].

We have implemented some simple data replication and migration policies. Any volume can be defined to be *mobile*, *shared*, or *stationary*. If the owner wants the data in a volume to follow him everywhere, he defines it mobile, and Segank attempts to propagate the data to as many devices as possible. To indicate that the volume may be accessed by others, the owner can declare it shared, in which case Segank attempts to propagate the data to a well-connected device and to cache a copy on the MOAD if possible. If it is defined stationary, the data is most likely to be needed only on this creator device, so no automatic propagation is attempted.

Connectivity A responsibility of the connectivity layer is to route to the MOAD regardless where it is and what physical communication interface it uses. A MOAD can be reached via: (1) an ad hoc wireless network encompassing both the requester and the target MOAD, (2) the Internet which connects to a remote ad hoc network within which the target MOAD is currently located, or (3) the target MOAD's cellular modem interface. For the first two cases, we use a combination of an ad hoc routing mechanism [17] and "Mobile IP." For point-to-point communication during Segankasts, the implementation supports TCP and UDP sockets and SUN-style RPCs. The experimental results are based on TCP sockets.

8 Experimental Results

8.1 Segankast Performance

We now study the performance of reading data using Segankast . In the first scenario, the user is at work accessing data on a mobile device. The mobile device is connected to the wired network through an ad hoc 802.11 link. The user owns three other devices that are located in his office LAN, and eight other devices located at five different cities that are at varying distances from his current location. The mobile device along with the eleven other storage devices comprise the personal storage system for the user. We will refer to this scenario as `WiFi-Work`. In the second scenario (referred to as `DSL-Home`), the user comes home and connects to the Internet using a DSL connection. The eleven other devices are accessible over the DSL connection.

The mobile device we use in our experiments is a Dell Inspiron Laptop with a 650 MHz, Intel P3 processor, 256 MB RAM, and a 10 GB, IBM Travelstar 20GN disk. The remaining devices are PlanetLab [19] nodes. Table 1 lists for each data source, the average latency for the reader to access a single 4 KB block from the source and the average bandwidth attained by the reader in accessing a stream of blocks from the same source under the two usage scenarios.

Figure 3 shows the trees determined by our tree building algorithm for optimizing latency and bandwidth in the `WiFi-Work` scenario.

We evaluate Segankast by running two experiments. The first measures the performance of reading small and

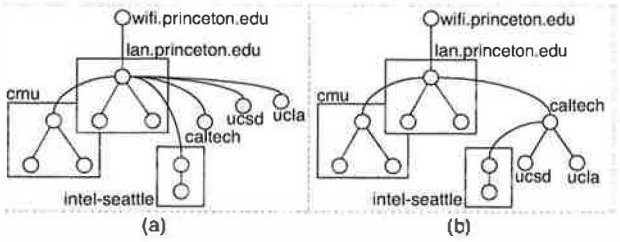


Figure 3: Segankast trees. (a) The tree for optimizing latency, (b) the tree for optimizing bandwidth.

Data Source	WiFi-Work		DSL-Home	
	Lat	Bw	Lat	Bw
princeton	15	0.49	140	0.068
cmu	45	0.42	160	0.065
intel-seattle	157	0.32	189	0.065
caltech	160	0.37	198	0.063
ucsd	173	0.36	223	0.063
ucla	177	0.32	243	0.062

Table 1: Latency (ms) and bandwidth (MB/s) of accessing data blocks directly from devices at different locations.

large files. The dataset consists of 25,000 small files each of size 4 KB and a single large file of size 50 MB. The files are initially created at the `ucla` device, which is farthest from the reader. We measure the performance of Segankast under the following settings. (1) Remote: only the `ucla` device has all the data, (2) Nearby: in addition to `ucla`, one of the nearby nodes at `lan.princeton.edu` has all of the files, and (3) Random: `ucla` has data, and each one of the files is replicated at three other randomly chosen locations.

For the second experiment, we use a disk trace collected on a workstation running Windows 2000. During the monitoring process, the user performed activities that are typical to personal computer users, such as reading email, web browsing, document editing, and playing multimedia files. We use a portion of this trace comprising of 787,175 I/O requests accessing a total of about 9.2 GB of data. We execute the first 760,000 requests on eleven devices in a round-robin fashion with a switch granularity of 20,000 requests. We then measure the performance of the read operations contained in the last 27,175 requests by executing them on the twelfth device that is connected to the remaining devices using a 802.11 connection (the `WiFi-Work` scenario) or a DSL connection (the `DSL-Home` scenario).

For both experiments, we also measure the cost of reading the files in the presence of an Oracle that provides the location of the closest device that contains a replica of the desired data. This allows us to bound the performance of alternative mechanisms such as location maps that track the location of the objects in the system. Since the costs of maintaining and querying the location map are not included in our Oracle mechanism, the resulting read performance is an upper-bound on the performance of an implementation that uses location maps.

Data Layout	Segankast		Oracle	
	sread (s)	lread (s)	sread (s)	lread (s)
Remote	563	132	552	122
Nearby	76	109	74	108
Random	234	107	226	105

Table 2: Read performance for the `WiFi-Work` scenario. `sread` refers to reading the small files. `lread` is the large file read.

Data Layout	Segankast		Oracle	
	sread (s)	lread (s)	sread (s)	lread (s)
Remote	1266	711	1096	702
Nearby	781	689	778	687
Random	1126	693	1019	688

Table 3: Read performance for the `DSL-Home` scenario.

The results of the two experiments are shown in Tables 2, 3, and 4. The performance measurements reveal that the overhead introduced by Segankast is minimal for reading large files even when the file is fetched from a remote `ucla` node through two intermediate hops. The cost of routing data through higher latency overlay paths and the overheads of forwarding requests and replies are amortized by pipelining the block fetches. In the `WiFi-Work` setting, the cost of performing small file reads is only marginally worse than a direct fetch. However, in the `DSL-Home` setting, the tree designed to optimize latency has some overlay paths that are more expensive than the direct connections and the reads incur a 10-15% overhead when a copy of the required data is not available at a nearby `lan.princeton.edu` device.

8.2 Sharing Experiments

We now examine the performance of Segank when users share data. In the first set of experiments, the setup consists of a writer node sharing a Segank volume with a reader node. We measure the performance of the reader as the writer creates new data and exports a new read-only snapshot to the reader. We define the "snapshot refresh latency" at the reader to be the time difference between the reader first expressing the desire to read and performing the actual read on a snapshot. A key result is that the refresh latency is solely a function of the network latency between the writer and the reader. In particular, it is independent of the amount of new data created by the writer in the new snapshot. We perform experiments where we vary (1) the type of the network link (Table 5), and (2) the amount of new data written by the writer before creating the new snapshot (Table 6).

Each experiment proceeds in three phases. The first

Scenario	Segankast	Oracle
WiFi-Work	499	496
DSL-Home	1763	1714

Table 4: Execution time in seconds of a trace collected on a personal computer.

	LAN	WiFi	WAN	Modem
BW (MB/s)	11	0.5	2.2	0.01
RTT (ms)	0.2	1.4	10	110

Table 5: Bandwidth and latency of the network links used. LAN refers to a 100 Mb/s Ethernet, WiFi to wireless 802.11 cards in the ad-hoc mode at 11 Mb/s bit-rate, WAN to a wired connection between Princeton and Yale, and Modem to a dial-up modem connection.

phase initializes the Segank volume at the reader. In the second phase, the writer performs a number of updates and then, creates a new read-only snapshot. In the final phase, the reader performs a "refresh" operation to express its desire to read from the new snapshot. Subsequently, a read-benchmark is run at the reader to read data from the new snapshot.

In the initialization phase, two identical, but distinct, directory trees are created. We name them T_1 and T_2. Each tree is 5 levels deep, where each non-leaf directory contains 5 sub-directories. In each directory, 10 files are created, each of size 8 KB. Thus, each tree has a total of 781 directories and 7,810 files, comprising about 64 MB of data. The data and invalidation log created during the initialization are at the reader only.

The update phase at the writer overwrites all files in the first K levels in T_1. We perform three sets of experiments with K taking values 3, 4 and 5. We name these experiments "Small", "Medium" and "Large".

The read-benchmark run at the reader involves reading 1,000 randomly-selected files from either T_1 or T_2. Thus, it reads about 7.8 MB of file data, in addition to reading some directory data and meta-data. In the rest of this section, "reading" a particular tree means running the read-benchmark for the given tree at the reader.

The left portion of Table 7 summarizes the results from this set of experiments. The refresh latency (column 3) is observed to be determined only by the network latency between the reader and the writer, and not by the amount of new data or invalidation log created in the new snapshot. As shown in column 9, complete data propagation can take several tens of seconds or even minutes on slow networks. Thus, compared to alternatives which mandate complete data propagation and replay before allowing access to new data, Segank incurs significantly lower user-perceived latency.

The columns labeled "Read Time" are listed to show

Experiment	Data (MB)	Log (KB)
Small	2.6	5.2
Medium	12.9	25.8
Large	62.1	124.1

Table 6: Amount of data and invalidation log generated at the writer in experiments of each type.

the overhead of lack of invalidation log propagation on read performance of the reader. The base case, labeled "LL" (for "Local" log and "Local" data), is the time for reading any of the two trees immediately after the initialization phase. Here, the complete invalidation log and all the data to be read are present locally at the reader. The column labeled "DD" (for "Distributed" log and "Distributed" data) lists the time for reading the updated tree T_1 immediately after the refresh operation. In this case, the reader neither has the complete log, nor all the data. The "DL" case is for reading the non-updated tree T_2 immediately after the refresh operation. The "LD" case is for reading T_1 after the new log (but not the new data) has been propagated from the writer to the reader.

Comparing LL with DL, and LD with DD, we conclude that not propagating the log adds significant overhead to the read performance for all network types except the LAN. When the complete log is present locally, network communication is needed only when the required data is not present locally. When the complete log is not present locally, each read request invariably results in network communication. As column 8 illustrates, the log propagation can usually be achieved in significantly less time than data propagation. This validates Segank's default policy of being highly aggressive in propagating the log, although the propagation is only performed in the background.

A second set of experiments is performed to evaluate Segank in a scenario where multiple users write to the same Segank volume concurrently. The setup consists of two nodes A and B. As in the single-writer case above, each experiment here proceeds in three phases. In the first phase, the volume is initialized to contain two trees T_1 and T_2. The complete log and the data is propagated to both A and B. Then, in the second phase, both A and B write to the volume concurrently. Node A overwrites files in the first few levels of T_1 while B does the same in T_2. To allow this, the system creates two snapshots, one for each writer. During concurrent writing, new updates performed by a node are visible only to that node. In phase three, the two nodes decide to merge their private snapshots. After the merge, each writer is able to read the new data created by the other in the second phase.

In our experiments, nodes A and B update the volume in non-conflicting ways. So, the system is able to merge the two snapshots automatically. We define "merge la-

Link Type	Expt. Type	Refresh (ms)	Read Sharing: Read Time (s) LL	DD	DL	LD	Log-Prop. Time (s)	Data-Prop. Time (s)	Write Sharing: Merge (ms)	Read Time (s) DD	DL
LAN	Small	0.4	16	17	17	18	0.002	2	0.8	18	17
	Medium			16	17	16	0.004	3		15	16
	Large			21	16	21	0.014	8		19	23
WiFi	Small	2.9	16	27	25	19	0.014	6	6.0	29	30
	Medium			29	25	21	0.049	26		34	32
	Large			44	25	42	0.236	119		45	35
WAN	Small	20	16	70	69	18	0.042	2	40	72	70
	Medium			72	69	28	0.086	6		70	71
	Large			78	69	68	0.131	43		79	79
Modem	Small	130	16	465	393	83	0.63	273	355	473	415
	Medium			664	398	347	2.6	1356		643	394
	Large			1306	403	1212	12.3	6478		1312	402

Table 7: Timing results (median from 3 runs) for the read-sharing and the write-sharing experiments.

tency" as the difference between the time when the two users express the desire to merge their versions, and the time when the system has successfully merged both versions so that the users can see each other's updates.

Results from this set of experiments are presented in the right portion of Table 7. The merge latency (column 10) in this case is observed to be independent of the amount of data. Note that the merge latency includes the time it takes for the system to detect if there are any conflicts between the two versions. Since in our experiments, updates of A and B are restricted to separate directories, it takes only a constant number of network messages to detect that there are no conflicts. Segank does not require any log or data propagation during automatic merging, unless there are conflicts in which case user intervention or application-specific conflict resolvers are needed.

Columns labeled "DL" and "DD" are times for reading T_1 and T_2 at A after the merge. Note that A does not have the complete log after the merge, so all reads result in network communication. These columns are similar to the "DL" and "DD" columns for the read-sharing case. The results for reading the trees at node B after merge are similar, and therefore, omitted.

9 Related Work

In addition to the issues already explored about Bayou [18, 22] in Section 2.2, another important difference is that Bayou is an application construction framework designed for application-specific merging and conflict resolution, while Segank is a storage/file system level solution. Segank allows many existing applications to run transparently, but it provides little support for merging and conflict resolution. An ongoing research topic is to investigate how the techniques that Segank employs to exploit a non-uniform network can be applied to a Bayou-like system to eliminate some of its limitations (such as its requiring full replicas).

Ivy [14] is a DHT-based file system. Both Segank and Ivy query logs of multiple users and use snapshots to support sharing. Segank's logs contain only object invalidation records, while Ivy's logs contain NFS operations and their associated data. Playing the invalidation logs and creating snapshots in Segank are light weight. Ivy effectively stores data twice, once in its NFS operation logs, and once again when the logs are played to create snapshots. Segank allows more flexible data placement than Ivy's DHT-based approach. This flexibility is essential for the non-uniform network that it targets.

Fluid Replication [10], an extension based on Coda, introduces an intermediate level between mobile clients and their stationary servers, called "WayStations," which are designed to provide a degree of data reliability while minimizing the communication across the wide-area used for maintaining replica consistency. Fluid Replication represents a middle point between Segank and Coda, in that it allows clients to interact with each other via WayStations without requiring them to connect to a far-away server. However, clients do not communicate with each other directly under Fluid Replication.

Distributed databases [7, 15], like Bayou, use update logs to keep replicas consistent. The purpose of the invalidation logs in Segank is not to replicate data. The invalidation log in Segank contains only invalidation records and the system does not need to propagate data to quickly bring other devices up-to-date.

JetFile uses multicasts to perform "best-effort" invalidation of obsolete data [8]. Instead of using a "symmetric" solution for reads and writes, Segank uses an "asymmetric" solution: Segankast for reads and log-based lazy invalidation for writes. We believe that this delayed and batched propagation of invalidation records is more appropriate for a more dynamic and non-uniform network.

Although we have called Segankast a "multicast-like" solution, it is actually quite different from overlay multicast systems [9, 4]. Typically, the goal of existing multicast systems is to deliver data to all machines in the target set. In contrast, the goal of a Segankast is to retrieve a single copy from several possible locations: the Segankast request need not always reach all possible locations, and one or more data replies may return.

The PersonalRAID system [21] is designed to manage a *disconnected* set of devices, which forces it to maintain complete replicas at all devices (except the mobile device). It is primarily a single user system with no support for data sharing among multiple users.

10 Conclusion

We have constructed a mobile storage system that is designed to manage storage elements distributed over a non-uniform network. Like some systems for a wired network, it needs to allow flexible placement and consistent access of distributed data. Like systems designed for disconnected operation and/or weak connectivity, it needs to avoid over-using weak links. The Segank system must account for a possible simultaneous coexistence of a continuum of connectivity conditions in the mechanisms that the system uses to locate data, to keep data consistent, and to manage sharing. It allows a mobile storage user to make the most of what a non-uniform network has to offer without penalizing him with unnecessary foreground propagation costs.

References

[1] ANDERSON, T., DAHLIN, M., NEEFE, J., PATTERSON, D., ROSELLI, D., AND WANG, R. Serverless Network File Systems. *ACM Transactions on Computer Systems 14*, 1 (1996), 41–79.

[2] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-Area Cooperative Storage with CFS. In *Proceedings of the ACM Eighteenth Symposium on Operating Systems Principles* (October 2001), pp. 202–215.

[3] DE JONGE, W., KAASHOEK, M. F., AND HSIEH, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (December 1993), pp. 15–28.

[4] DEERING, S. E., ESTRIN, D., FARINACCI, D., JACOBSON, V., LIU, C.-G., AND WEI, L. An Architecture for Wide-Area Multicast Routing. In *Proc. of SIGCOMM'94* (London, UK, August 1994), pp. 126–135.

[5] GARG, N., SHAO, Y., ZISKIND, E., SOBTI, S., ZHENG, F., LAI, J., KRISHNAMURTHY, A., AND WANG, R. Y. A Peer-to-Peer Mobile Storage System. Tech. Rep. TR-664-02, Computer Science Department, Princeton University, October 2002.

[6] Gnutella. http://gnutella.wego.com/.

[7] GORELIK, A., WANG, Y., AND DEPPE, M. Sybase Replication Server. In *Proc. ACM SIGMOD Conference* (May 1994), p. 468.

[8] GRONVALL, B., WESTERLUND, A., AND PINK, S. The design of a multicast-based distributed file system. In *Operating Systems Design and Implementation* (1999), pp. 251–264.

[9] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND O'TOOLE, J. W. Overcast: Reliable Multicasting with an Overlay Network. In *Proc. the Fourth Symposium on Operating Systems Design and Implementation* (October 2000).

[10] KIM, M., COX, L. P., AND NOBLE, B. D. Safety, Visibility, and Performance in a Wide-Area File System. In *Proc. First Conference on File and Storage Technologies* (January 2002).

[11] KISTLER, J., AND SATYANARAYANAN, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 3–25.

[12] LEE, E. K., AND THEKKATH, C. E. Petal: Distributed Virtual Disks. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996), pp. 84–92.

[13] MUMMERT, L. B., EBLING, M. R., AND SATYANARAYANAN, M. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles* (December 1995).

[14] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: A Read/Write Peer-to-Peer File System. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (December 2002).

[15] ORACLE CORPORATION. *Oracle7 Server Distributed Systems: Replicated Data*, 1994.

[16] PEASE, D. A., MENON, J., REES, B., DUYANOVICH, L. M., AND HILLSBERG, B. L. IBM Storage Tank - A heterogeneous scalable SAN file system. *IBM Systems Journal 42*, 2 (2003), 250–67.

[17] PERKINS, C., BELDING-ROYER, E., AND DAS, S. Ad Hoc On Demand Distance Vector (AODV) Routing, 2001.

[18] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. ACM Symposium on Operating Systems Principles* (1997).

[19] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. First Workshop on Hot Topics in Networks (HotNets-I)* (October 2002).

[20] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the ACM Eighteenth Symposium on Operating Systems Principles* (October 2001).

[21] SOBTI, S., GARG, N., ZHANG, C., YU, X., KRISHNAMURTHY, A., AND WANG, R. Y. PersonalRAID: Mobile Storage for Distributed and Disconnected Computers. In *Proc. First Conference on File and Storage Technologies* (January 2002).

[22] TERRY, D. B., THEIMER, M. M., PETERSON, K., DEMERS, A. J., SPREITZER, M. J., AND HAUASER, C. H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 172–183.

[23] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: A Scalable Distributed File System. In *Proc. ACM Symposium on Operating Systems Principles* (1997).

[24] WAGNER, J. Getting to Know Your 3G. http://www.internetnews.com/wireless/article/-0,,10692_964581,00.html, January 2002.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

SAGE

SAGE is a Special Technical Group (STG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

❖ Ajava Systems, Inc. ❖ Aptitune Corporation ❖ Atos Origin B.V. ❖

❖ Computer Measurement Group ❖ Delmar Learning ❖

❖ DoCoMo Communications Laboratories USA ❖

❖ Electronic Frontier Foundation ❖ Interhack Corporation ❖

❖ MacConnection ❖ The Measurement Factory ❖ Microsoft Research ❖

❖ Portlock Software ❖ Sun Microsystems, Inc. ❖ Taos — The Sys Admin Company ❖

❖ UUNET Technologies, Inc. ❖ Veritas Software ❖

SAGE Supporting Members

❖ Ajava Systems, Inc. ❖ Microsoft Research ❖

❖ MSB Associates ❖ Ripe NCC ❖

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-18-8